A Überblick über die 11. Übung

- Einführung in das .NET Framework
- Einführung in C#, speziell Unterschiede zu Java

A.1 Was ist .NET?

- CLI: *Common Language Infrastructure*
  - legt die Infrastruktur fest, um .NET Anwendungen zu verwenden
  - CTS und CLR sind Teile der CLI
- CTS: *Common Type System*
  - Definiert die gemeinsamen Typen
- CLS: *Common Language System*
  - Untermenge des CTS
  - Definiert die minimalen Anforderungen an die Programmiersprache

A.1 Was ist .NET? (2)

- CLR: *Common Language Runtime*
  - Implementierung des CTS
  - Definiert die gemeinsamen Typen
  - Ausführungsumgebung (VM) für verwaltete Programme (managed code)
  - sicheres Laden und Ausführen
  - Garbage Collection
  - Sicherheitsüberprüfungen
  - Klassenbibliothek

A.2 .NET im CIP-Pool

- Mono von Ximian
  - Open-Source Implementierung des .NET Development Framework
  - unterstützt Linux, Unix und Windows
  - JIT für x86-Architektur, Interpreter für s390, SPARC und PowerPC
- Linuxversion im CIP-Pool unter: **`/local/mono`**
  - **`mcs`**: C#-Compiler
  - **`mono`**: CLI Ausführungsumgebung
- Testen:

```
> setenv PATH /local/mono/bin:$PATH
> cd /proj/i4mw/pub/net
> mcs Hello.cs
Compilation succeeded
> mono Hello.exe
```

A.2 Beispiel: BankLibrary

- Die Bibliothek: Bank.cs

```
namespace BankLibrary {

    public class Bank {
        public static int deposit (Account account,
                                   int amount){
            return account.deposit(amount);
        }
    }

    public class Account {
        private int value = 0;
        public int deposit( int amount ) {
            value += amount;
            return value;
        }
```

- Bibliothek erstellen (erzeugt wird: **`Bank.dll`**)

```
> mcs -target:library Bank.cs
```


A.2 Beispiel: BankClient

- Anwendung: Customer.cs

```
using System;
namespace Customer {

    class Customer {
        public static void Main(string[] args) {
            BankLibrary.Account account =
                new BankLibrary.Account();
            Console.WriteLine("deposit 3 -> account = {0}",
                BankLibrary.Bank.deposit(account, 3));
            Console.WriteLine("deposit 5 -> account = {0}",
                BankLibrary.Bank.deposit(account, 5));
        }
    }
```

- Anwendung erstellen (erzeugt wird: **`Customer.exe`**)

```
> mcs -r:Bank.dll Customer.cs
```

- Anwendung starten

```
> mono Customer.exe
```


B C# vs. Java

B.1 C# - Überblick

- Streng typisierte objekt-orientierte Programmiersprache
- wird übersetzt in *Intermediate Language* (IL): ähnlich Java-Bytecode
- wird ausgeführt von CLR - ähnlich JVM
- Anforderungen:
  - Architekturunabhängigkeit
  - Sprachunabhängigkeit!
- Vorbilder: Java und C++
- ECMA Standard 334

B.2 Gemeinsamkeiten von C# und Java

- keine Header-Dateien
- Mehrfachvererbung von Schnittstellen (nicht von Implementierungen)
- keine globalen Funktionen oder Konstanten (alles in Klassen)
- Arrays und Strings mit festen Längen und Zugriffskontrolle
- Alle Variablen müssen vor der ersten Verwendung initialisiert werden
- alle Objekte erben von **`Object`**-Klasse
- Objekte werden auf dem Heap erzeugt (mit dem Schlüsselwort **`new`**)
- Garbage Collector
- Thread Unterstützung, Synchronisation
- Reflection

B.3 Erste Schritte

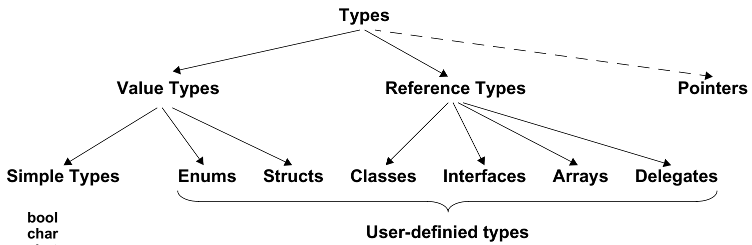
- Hello World

```
using System;

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

- alle Klassen der .NET-Architektur im `System`-Namespace (`using`-Anweisung)
- Main-Methode als Einstiegspunkt:
  `public static void Main(string[] args) { ... }`

B.4 Typsystem



bool
char
sbyte
byte
short
ushort
int
uint
long
ulong
float
double
decimal

- Value Types
  - werden auf dem Stack erzeugt
  - bei einer Zuweisung wird der Wert kopiert
- Reference Types
  - werden auf dem Heap erzeugt
  - bei einer Zuweisung wird die Referenz kopiert

1 Primitive Typen

- `sbyte`, `short`, `int`, `long`, `float`, `double`, `bool`, `char` (Unicode)
  - wie in Java
- vorzeichenlose Typen: `byte`, `ushort`, `uint`, `ulong`
- `decimal`: 128 Bit Zahl
- Konstanten: wie in C/C++

```
const int var = 3;
```

2 Enums

- Liste von Konstanten mit Namen

```
public enum Farbe { Gelb=1, Rot=2, Gruen=4 };

Farbe farbe = Farbe.Rot | Farbe.Gelb;
if ((farbe & Farbe.Rot) != 0) { ... }
```


3 Arrays

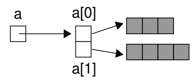
- Eindimensionale Arrays: wie in Java

```
int[] array = new int[3];
int[] array = new int[] { 1, 2, 3 };
int[] array = { 1, 2, 3 };

int len = array.Length;
```

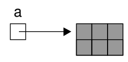
- Mehrdimensionale Arrays
  - "Jagged": Arrayelemente zeigen auf weitere Arrays

```
int[][] 2D_array = new int[2][];
a[0] = new int[3]; a[1] = new int[4];
```



  - "Rectangular": Abbildung auf eindimensionales Array

```
int[,] 2D_array = new int[2,3];
len = 2D_array.Length;         // 6
len = 2D_array.GetLength(0);   // 2
len = 2D_array.GetLength(1);   // 3
```



4 Strings

- Objekte der Klasse **`string`** sind konstante, invariante Objekte
  - ◆ bei Modifikationen wird neues **`string`**-Objekt als Rückgabewert geliefert
  - ◆ Aneinanderhängen mit "+"
  - ◆ Indizierung: **`s[i]`**, Länge der Zeichenkette: **`s.Length`**
- Vergleich von **`string`**-Objekten

```
string s1 = "Hello";
string s2 = string.Copy(s1);

if (s1 == s2) { ... }                  // true (1)
if ((Object)s1 == (Object)s2) { ... }  // false (2)
```

  - ◆ (1) Die Klasse **`string`** überlädt den ==-Operator:
    Vergleich der Zeichenketten (string ist *reference type*!)
  - ◆ (2) ==-Operator der **`Object`**-Klasse: Vergleich von Referenzen
  - ◆ (3)Vergleich der Zeichenketten

4 Strings

- Strings parsen

```
char[] trennzeichen = {' ', ',', ':'};
// oder:
string s_trennzeichen = " ,:";
char[] trennzeichen = s_trennzeichen.ToCharArray();

string test = "Vorname,Nachnahme Strasse:PLZ,Ort";
```

- Ein-/Ausgabe (Console)

```
using System;

Console.Write("kein Newline am Ende");
Console.WriteLine("diesmal mit Newline");
Console.WriteLine("Heute ist der {0}. {1}", 22, "Januar");
```


5 Strukturen

- C# unterstützt Strukturen:

```
public struct Account {
    public int balance;
    public Account(int amount) { balance = amount; }
    public void Withdraw(int amount) { balance -= amount; }
}
```

  - ◆ ähnlich C/C++, aber auch Konstruktoren und Methoden möglich
- Strukturen werden in C# auf dem Stack angelegt (*value type*)
  - ◆ Klassen (**`class`**) werden immer auf dem Heap erzeugt
- Keine Vererbung möglich, Strukturen können aber Interfaces implementieren

6 Typunifizierung / Boxing

- Value types (primitive Type, Strukturen, Enums) können in ein Objekt (*Boxing*) und wieder zurück gewandelt werden (*Unboxing*):

```
Object obj = 333;            // boxing (explizit)
int i = (int) obj;           // unboxing

Stack stack = new Stack();   // Stack enthält Objekte
stack.Push (i);              // boxing (implizit)
```

- Nachteile:
  - ◆ Wrapper-Objekt muss (auf dem Heap) erzeugt werden
  - ◆ Man erkennt nicht auf Anhieb, dass das Verfahren teuer ist
- alle primitiven Typen sind als Strukturen definiert
  - ◆ z.B. **`int`** als Alias für **`System.Int32`**
  - ◆ definiert als "**`public struct Int32 { ... }`**"
  - ◆ spezielle Behandlung durch den Compiler

7 Zeigerarithmetik

- Zeiger können in C# verwendet werden
  - wie in C/C++

```
int i = 5;

unsafe void inc(int *i) { *i = *i + 1; }

inc(&i);
```

- Methoden, die Zeiger enthalten, müssen mit **`unsafe`** markiert werden
  - Diese Kennzeichnung soll den Gebrauch von Zeigern minimieren
  - Der Garbage Collector ignoriert die entsprechenden Objekte
  - kann diese aber immer noch im Speicher verschieben (beim Kompaktifizieren)
- Schlüsselwort **`fixed`**
  - Objekt wird im Speicher "gepinnt": kann nicht mehr verschoben werden

B.5 Anweisungen

1 Switch-Anweisung

- Kontrollfluss muss explizit festgelegt werden
  - **`break`** (oder **`return`**, **`goto`**, **`throw`**) muss am Ende jeder **`case`**-Anweisung stehen
  - falls kein **`case`** zutrifft: **`default`**-Label
- als Switch-Typ ist auch ein String erlaubt:

```
switch(name) {
    case "Zaphod Beeblebrox":
        Console.WriteLine( "Hello Zaphod" );
        break;
    case "Ford Prefect":
        Console.WriteLine( "Hi" );
        break;
```


2 Collections, foreach-Anweisung

- In Java oder C++:

```
while (! collection.isEmpty()) {
    Object o = collection.get();
    collection.next();
    ...
}
```

- C#:

```
foreach (Object o in collection) { ... }
foreach (int i in array) { ... }
```

  - **`foreach`**-Anweisung arbeitet auf allen Objekten, die das Interface **`System.Collections.IEnumerable`** implementieren.
  - Auch Arrays implementieren dieses Interface

B.6 Assemblies, Namespaces und Zugriffslevel

- Namensräume (namespaces) wie in Java: Trennung mit "**`.`**"
  - "syntactic sugar" für lange Klassennamen
  - Namensräume importieren mit **`using`**-Schlüsselwort
- Eine Assembly besteht aus mehreren Dateien (einem Projekt), die zu einer .exe- (Executable) oder .dll-Datei (Bibliothek) kompiliert werden
  - definieren einen eigenen Namensraum
  - verschiedene Versionen einer Assembly können parallel existieren
- Fünf Zugriffslevel:
  - **`private`** (Zugriff nur innerhalb der Klasse, wie in Java)
  - **`internal`** (Zugriff innerhalb der Assembly)
  - **`protected`** (Zugriff innerhalb der Klasse und abgeleiteter Klassen)
  - **`internal protected`** (wie **`protected`**, zusätzlich im Assembly)
  - **`public`** (Zugriff immer erlaubt)

B.7 Methoden

1 Aufrufsemantik

- Standardmäßig wird “call by value” verwendet
- **`out`**-Parameter: “call by reference”
  - ◆ übergebene Variable braucht nicht initialisiert worden sein
  - ◆ schreibender Zugriff auf Variablen des Aufrufers notwendig

```
class Test
{
    public static void Main() {
        int zufallszahl;
        random (out zufallszahl);
        ...
    }

    public static void random (out int r) {
        r = ...;
```


1 Aufrufsemantik

- **`ref`**-Parameter: ebenfalls “call by reference”
  - ◆ übergebene Variable muss zuvor vom Aufrufer initialisiert worden sein (“in-out”-Parameter)

```
class Test
{
    public static void Main() {
        int a = 1, b = 2;
        swap (ref a, ref b);
    }

    public static void swap (ref int a, ref int b) {
        int temp = a;
        a = b;
        b = temp;
```

- in Java kein “call by reference” für primitive Typen möglich (Verwendung einer Holder-Klasse (s. CORBA) oder eines Arrays notwendig)

2 Beliebige Anzahl von Parametern

- Das Schlüsselwort **`params`** ermöglicht eine variable Anzahl von Parametern:

```
class Test
{
    public static void Main() {
        int ergebnis = add(1, 2, 3, 4);
        ...
    }

    public static int add (params int[] array) {
        int sum = 0;
        foreach (int i in array)
            sum += i;
        return sum;
```


3 Überladen von Operatoren

- Beispiel:

```
class Score : IComparable {
    int value;

    public static bool operator == (Score x, Score y) {
        return x.value == y.value;
    }
    public static bool operator != (Score x, Score y) {
        return x.value != y.value;
    }
    public int CompareTo (Object o) {
        return value - ((Score)o).value;
    }
}

Score a = new Score (5);
Score b = new Score (5);
Object c = a, d = b;

if (a == b) {...}                              // true
```

4 Properties

- Properties werden wie Variablen angesprochen
  - Zugriffe werden in Methodenaufrufe von Get- und Set-Funktionen umgesetzt
  - "virtuelle Variablen": existieren nicht, werden semantisch vorgetäuscht

```
class C {
   private string s = "(not defined)";

   public string S {
      get { return this.s; }
      set {
         if (value == null) throw new Exception("null");
         this.s = value.ToUpper();
      }
   }
   public static void Main(string[] args) {
      C c = new C();
      c.S = "Hello World";
```

  - **get** oder **set** kann auch weggelassen werden, die Operation ist dann nicht erlaubt

4 Properties

- Der Compiler erzeugt daraus die beiden folgenden Methoden:

```
public string get_S() { return this.s; }
public void set_S(string value) {
    if (value == null)
        throw new Exception("null");
    this.s = value.ToUpper();
}

C c = new C();
```

- Innerhalb des **set**-Abschnitts kann der Parameter, welcher der Set-Methode übergeben wird, als **value** angesprochen werden
- Vorteil: bessere Lesbarkeit

```
// Java:
window.setSize (getSize() + 1);
// C#:
window.size++;
```


4 Properties: Indexer

- Objekte wie Arrays behandeln
- jedes Element wird über get/set-Methoden angesprochen

```
class Skyscraper {
    Story[] stories;

    public Story this [int index] {
        get { return stories [index]; }
        set {
            if (value != null)
                stories [index] = value;
        }
    }
    ...
}
```


5 Konstruktoren & Destruktoren

- Konstruktoren wie in Java
  - statische Konstruktoren möglich ("**static Klassenname()**")
  - Überladen von Konstruktoren möglich
- Destruktoren werden vom Garbage Collector aufgerufen (kein delete!)
  - Zeitpunkt des Aufrufs kann nicht vorhergesagt werden
  - Es ist nicht garantiert, dass der Destruktor überhaupt aufgerufen wird
  - **dispose()**-Methode für Aufräumarbeiten empfohlen (**IDisposable**), s. Garbage Collector

6 Events und Delegates

- Delegates sind eine Art typsicherer objekt-orientierter Funktionszeiger
- können mehrere Methoden (Handler) enthalten
- Sprachunterstützung für Event-Verarbeitung:

```
// Typ-Definition
delegate void MouseEventHandler(int x, int y);

// Event-Erzeuger
class Window {
    public event MouseEventHandler mouseChange;
}

// Event-Verbraucher
class MouseControl {
    public MouseControl(Window window) {
        window.mouseChange +=
            new MouseEventHandler(myhandler);
    }
    private void myhandler(int x, int y) {...}
```


B.8 Klassen, Interfaces, Vererbung

1 Syntax

- Übernommen von C++; Syntax identisch für Klassen und Interfaces

```
class D : B, C {...}
```

- Jedoch keine Vererbung unter Angabe von Zugriffsrechten
- Eine Klasse kann max. von *einer* anderen Klasse erben, aber *mehrere* Interfaces implementieren

2 Die Klasse `Object`

- Wurzelklasse, von der alle anderen Klassen abgeleitet sind

```
class C { ... }
// bedeutet implizit
class C : Object { ... }
```

- Mit `GetType()` kann der Typ der Klasse abgefragt werden (Reflection)

```
Object obj = new C();
Type type = obj.GetType ();
Console.WriteLine (type.Name);
```

- Virtuelle Methoden `ToString()` und `Equals()`

```
class Object {
    ...
    public virtual string ToString() { return GetType.Name; }
    public virtual bool Equals (Object other) {
        return this == other;
    }
```


3 Interfaces

- Explizite Implementierung eines Interface:

```
public interface ITeller {
    void Next();
}
public interface IIterator {
    void Next();
}

public class Clark : ITeller, IIterator {
    void ITeller.Next() { }
    void IIterator.Next() { }
```

- Vermeidung von Namenskonflikten bei mehreren Interfaces:

```
Clark clark = new Clark();
((ITeller)clark).Next();
```

- Eine Klasse kann nur von einer Basisklasse erben, aber mehrere Interfaces implementieren
  - ◆ Keine Vererbung bei Strukturen, aber Implementierung von Interfaces möglich

4 Typumwandlung (Cast)

- Besonderheiten / Beispiel:

```
class C {...}
class D {...}

Object obj  = new C();
Object obj2 = new D();

C c1 = (C) obj;                  // OK

C c2 = (C) obj2;                 // InvalidCastException
// da obj2 nicht auf ein C-Objekt verweist,
// wird eine InvalidCastException geworfen

C c3 = obj2 as C;                // Ergebnis: c3 = null
```

- Typ eines Objekts prüfen:

```
if (obj is C) {...}
```


5 Polymorphie und dynamisches Binden

- Beispiele

```
N n = new N ();
n.foo();            // foo der Klasse N
((D)n).foo();       // foo der Klasse D
((B)n).foo();       // foo der Klasse D (nicht B!)
```

- Späteres Update einer Klasse

```
class LibraryClass {  // Version 1
    public void CleanUp() { ... }
}
class PartitionMagic : LibraryClass {
    public void Delete() { ... Festplatte formatieren ... }
}

class LibraryClass {  // Version 2
    private string name;
    public virtual void Delete() { name = null; }
```

- In C++ würde `obj.CleanUp()` die Festplatte formatieren, in C# erzwingt der Compiler die Verwendung von `new` oder `override`

5 Polymorphie und dynamisches Binden

- In Java: alle Methoden sind virtuell
  In C#: Schlüsselwort `virtual` zur Kennzeichnung von virtuellen Methoden (wie in C++)

```
class B {
    public virtual void foo() { }
}
```

- Schlüsselwort `override` zum Überschreiben virtueller Methoden

```
class D : B {
    public override void foo() { }
}
```

- Schlüsselwort `new` zum Überschreiben nicht-virtueller Methoden

```
class N : D {
    public new void foo() { }
}
```


B.9 Reflection

- Zugriff auf Klasseninformationen über die `Type`-Klasse

```
Object obj = new C();
Type t = obj.GetType();  // Alternative 1: Object.GetType()
Type t = typeof (C);     // Alternative 2: typeof()
Type t = Type.GetType("C");     // Alt. 3: Type.GetType()
```

- Zugriff auf Metainformationen: `Type.GetConstructors()`, `Type.GetMethods()`, `Type.GetProperties()`, `Type.GetFields()`

- Methodenaufruf (Konstruktoraufruf analog)

```
MethodInfo m = t.GetMethod ("F", new Type[] { });
m.Invoke (obj, null);
```

- Properties
  - `Property.PropertyType`: Returncode (get) bzw. Parametertyp (set)
  - `Property.CanRead`, `Property.CanWrite`: Existiert get bzw. set?

B.10 Exceptions

- Beispiel

```
try {
    ...
} catch( Exception e ) {
    Console.WriteLine("{0}\n{1}", e.Message, e.StackTrace)
} finally {
    ...
```

- Die Signaturen der Methoden enthalten nicht die möglichen Exceptions!
- Oberklasse: `System.Exception`
  - Klasse `ApplicationException` für benutzerdefinierte Exceptions
- Abkürzung, wenn man keine Referenz auf die geworfene Exception braucht:

```
try { ... } catch(IOException) { ... }
// oder:
try { ... } catch { ... }
```


B.11 Attribute

- Erweiterung der vordefinierten Attribute `public`, `abstract` usw.
- zusätzliche Informationen (z.B. Zugriffsbeschränkungen)

```
[AuthorAttribute ("Bill Gates")]
class Windows
{
    [Localizable(true)]
```

  - bleibt im MSIL erhalten, kann zur Laufzeit ausgelesen werden
  - ähnlich “`/** */`” oder “`@tag`” in Java, die allerdings nicht in den Bytecode übernommen werden
- Beispiele
  - `[ConditionalAttribute]`
    bedinge Kompilierung; entspricht `#ifdef` von C/C++
  - `[Obsolete("this class is obolete", false)]`
    Compilerwarnung oder -fehler
  - `[Serializable]`: s. Übung zu Remoting

B.11 Attribute

- Mehrere Attribute gleichzeitig möglich

```
[Serializable][Obsolete]
class C { ...
```

- Verwendung von Parametern
  - Position bestimmt Parameter: `[Obsolete("Use class C instead")]`
  - Angabe des Parameternamens: `[Obsolete(IsError=true)]`
- Attribute müssen als Klassen definiert werden: Trace-Attribut als Beispiel

```
class TraceAttribute : Attribute {
    private int level = 0;
    public int Level {
        get { return this.level; }
        set { this.level = value; }
    }
}

class Window : IWindow {
    [Trace (Level = 1)]    // Abkürzung für "TraceAttribute"
```


B.11 Attribute

- Auslesen des Trace-Attributes (im Konstruktor von WindowProxy)

```
class WindowProxy : IWindow {
    private Window real;
    private Trace trace;

    public void Show() {
        if (trace.Level > 0)
            Console.WriteLine("--> Window.Show()");
        this.real.Show();
    }

    public WindowProxy(Window window) {
        real = window;
        Type type = typeof (Window);
        MethodInfo method = type.GetMethod
            ("Show", new Type[] {});
        Object[] attributes =
            method.GetCustomAttributes(true);
        foreach (Object attribute in attributes) {
            if (attribute.GetType() == typeof (Trace))
                Trace trace = (Trace) attribute;
```

B.12 Garbage Collector

- Ruft Destruktoren auf und gibt Speicher frei (nur reference types)
  - Sweepphase: Verfolgung aller erreichbaren Objekte
  - Finalisierungsthread (Destruktoren): parallel zum weiterlaufenden Prozess
- `GC.Collect()`
  - Sweepphase des Garbage Collectors erzwingen
  - wird bei Speichermangel automatisch aufgerufen
- `GC.KeepAlive(Object obj)`
  - Objekt von der Garbage Collection ausschliessen
  - falls unverwalteter Code, den der GC nicht “sieht”, auf das Objekt zugreift
- `GC.WaitForPendingFinalizers()`
  - wartet auf die Beendigung aller Finalisierungen
  - keine Garantie, dass diese Methode zurückkehrt (z.B. bei fehlerhaften Destruktoren)

B.12 Garbage Collector

- IDisposable Interface
  - Definition:

```
public interface IDisposable {
    void Dispose();
}
```

  - stellt eine Art expliziten Destruktor dar
  - Verwendung am besten mittels using-Konstrukt

```
public class aClass {
    public void aMethod() {
        ...
        using (ResourceWrapper rw = new RessourceWrapper()) {
            ... // use rw
        }
```

  - `rw.Dispose()` wird aufgerufen, wenn der `using`-Block verlassen wird oder wenn eine Exception auftritt

B.12 Garbage Collector

- Schwache Referenzen
  - `WeakReference r = new WeakReference(obj);`
  - Objekt, welches nur noch über schwache Referenz erreichbar ist, wird vom GC freigegeben (GC ignoriert schwache Referenzen)
  - nützlich falls Objekte in einer Art Cache gehalten werden
    (falls das Objekt sonst nicht mehr referenziert wird, soll auch die Referenz im Cache ignoriert werden)

B.13 Programmieren mit Threads

1 Erzeugen von Threads

- Thread-Klasse mit allen notwendigen Methoden:
  `Suspend()`, `Start()`, `Resume()`, `Sleep(int ms)`, `Join()`, `Abort()`

```
using System.Threading.Thread;

void MyThreadMethod()
public delegate void ThreadStart();
Thread thread = new Thread(new ThreadStart(MyThreadMethod));
```

  - `ThreadState` liefert den aktuellen Zustand des Threads:
    `Aborted`, `Running`, `Stopped`, `Suspended`, `Unstarted`

```
if (thread.ThreadState & ThreadState.Suspended) ...
```

  - Priorität gegenüber anderen Threads: `ThreadPriority`:
    `AboveNormal`, `BelowNormal`, `Highest`, `Lowest`, `Normal`

```
thread.Priority = ThreadPriority.Highest;
```

1 Erzeugen von Threads

- System stellt ThreadPool bereit (zum effizienten Erzeugen von Threads)

```
using System.Threading.ThreadPool;

void MyThreadMethod(); ...
ThreadPool.QueueUserWorkItem(
    new WaitCallback(MyThreadMethod), null);
```

2 Synchronisation

- Gesperrter Bereich (wie `synchronized(Object)` in Java)

```
Object obj = new Object();
lock(obj) { ... }
lock(typeof(Object)) { ... }
lock(this) { ... }
```

- kein `synchronized`-Schlüsselwort, aber Synchronized-Attribut

```
[MethodImpl(MethodImplOptions.Synchronized)]
public void SynchronizedMethod() { ... }
```


2 Synchronisation

- Klasse Monitor

```
Monitor.Enter(obj);                // wie lock(obj) {
Monitor.Exit(obj);                 // ... }
Monitor.TryEnter(obj, 1000);       // max. 1000 ms warten
while (!Monitor.TryEnter(obj))
    // z.B. Events der Anwendung verarbeiten
```

- Warten und Aufwecken (Java: `wait`/`notify`/`notifyAll`)

```
lock(obj) { ... Monitor.Wait(obj); ... }      // Thread A
...
lock(obj) { ... Monitor.Pulse(obj); ... }     // Thread B
lock(obj) { ... Monitor.PulseAll(obj); ... }
```


B.14 XML-Kommentare

- Ähnlicher Mechanismus wie Javadoc

```
/// ... comment
class C { ...
```

- Erzeugung des XML-Files

```
> mcs -doc:Test.xml Test.cs
```

- Vordefinierte XML-Tags:
  `<summary>` kurze Beschreibung `</summary>`
  `<remarks>` ausführliche Beschreibung `</remarks>`
  `<param name="Name">` Beschreibung eines Parameters `</param>`
  `<returns>` Beschreibung des Rückgabewerts `</returns>`

  Weitere Tags:
  `<exception>`, `<example>`, `<code>` usw.

B.15 Collections.ArrayList

- ähnlich Java-Vector

```
using System.Collections;

class Account {
    public string name;
    public int balance;
    public Account(string name) { this.name = name; }
}

class Bank {
    ArrayList al = new ArrayList();

    void addCustomer(string name) {
        al.Add(new Account(name));
    }
    void showAllCustomers() {
        IDictionaryEnumerator myEnum = al.GetEnumerator();
        while (myEnum.MoveNext()) {
            Account acc = (Account)myEnum.Current;
            Console.WriteLine("{0}: {1}",
```

B.16 Collections.Hashtable

- ähnlich Java-Hashtable

```
using System.Collections;

class Bank {
    Hashtable ht = new Hashtable();

    void addCustomer(string name) {
        ht.Add(name, new Account(name));  // key, value
    }
    void deposit(string name, int howmuch) {
        if (!ht.ContainsKey(name)) return;
        Account acc = ht[name]; // liefert null, wenn der
                                // Schlüssel nicht exist.
        acc.balance += howmuch;
    }
    void showAllCustomers() {
        IDictionaryEnumerator myEnum = ht.GetEnumerator();
        while (myEnum.MoveNext()) {
            Account acc = (Account)myEnum.Value;
            Console.WriteLine("{0}: {1}", myEnum.Key,
```