Übungen zu

Objektorientierte Konzepte in Verteilten Systemen und Betriebssystemen

Wintersemester 2002/03

Teil III: .NET

A Überblick über die 11. Übung

- Einführung in das .NET Framework
- C#: 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 Systems*
  - 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

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 für Schnittstellen (nicht für 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
- Verzicht auf Zeiger (C/C++)
- Bedingungen nur mit **`boolean`**-Werten

```
//kein
if ( ref ) { ... }                    // Compiler-Fehler
// stattdessen
if ( ref != null ) { ... }
```


B.3 Gemeinsamkeiten von C# und Java

- alle Objekte erben von **`Object`**-Klasse
- Objekte werden auf dem Heap erzeugt (mit dem Schlüsselwort **`new`**)
- Garbage Collector
- Thread Unterstützung (z.B.: **`synchronized`** - Konstrukt)
- Reflection

B.4 Assemblies, Namespaces und Zugriffslevel

- Namensräume (namespaces) wie in Java
  - "syntactic sugar" für lange Klassennamen
  - Namensräume importieren mit **`using`** Schlüsselwort
- Eine Assembly besteht aus mehreren Dateien (einem Projekt), die zu einer .exe oder .dll-Datei 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.5 Kleinigkeiten

- Konstanten
  - wie in C/C++:

```
//Java:
static final int var = 3;

//C#
const int var = 3;
```

- Exceptions:
  - Methoden Signaturen enthalten nicht die möglichen Exceptions
  - Oberklasse: **`System.Exception`**
  - Abkürzung, wenn man keine Referenz auf die geworfene Exception braucht:

```
try {
    //...
} catch {
    //...
}
```


B.6 Switch-Anweisung

- Kontrollfluss muss explizit festgelegt werden:

```
//Java:
int switchcase = 3;

switch(switchcase) {
    case 3:
        System.out.println( "World" );
    default:
        System.out.println( "Peace" );
}
```

- als Switch-Typ ist auch ein String erlaubt:

```
String name = "Zaphod Beeblebrox"

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


B.5 Kleinigkeiten (2)

- Vererbung
  - Syntax beim ableiten von Klassen bzw. implementieren von Interfaces wie in C++:
  - jedoch keine Vererbung unter Angabe von Zugriffsrechten

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

  - Standard-Klassenrecht: private
- Enums:

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

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

- 12 primitive Datentypen
  - wie in Java + zusätzliche unsigned-Typen
  - **`sbyte`**, **`byte`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`**, **`ulong`**, **`char`**, **`float`**, **`double`**,12-byte Fließkommazahl "**`decimal`**"

B.7 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
// obwohl obj nicht auf ein C-Objekt verweist,
// wird keine Exception ausgelöst
// stattdessen erhält c3 den Wert null
```

- Typ eines Objekts prüfen:

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

B.8 Properties (1)

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

```
using System;

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";
        Console.WriteLine(c.S);
    }
}
```


B.8 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;
        }
    }
    ...
}

Skyscraper empireState = new Skyscraper(...);
empireState[102] = new Story ("The Top One", ...);
```


B.8 Properties (2)

- Der Compiler erzeugt hieraus 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();
c.set_S("Hello World");
Console.WriteLine ( c.get_S() );
```

- 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++;
```


B.9 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;
    ...
    mouseChange(y, x);
    ...
}

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

B.10 Collections, foreach-Anweisung

- In Java oder C++:

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

for (int i = 0; i < array.length; i++) {...}
```

- 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.11 Strukturen

- C# unterstützt wie C/C++ Strukturen:

```
public struct Vector {
    public float direction;
    public int magnitude;
}

Vector[] vectors = new Vector[1000];
```

- Strukturen werden in C# jedoch immer auf dem Stack angelegt (Erweiterung der primitiven Datentypen)

B.12 Typunifizierung / Boxing (1)

- Einfache Typen und Strukturen sind Werttypen (*value types*)
- Klassen sind Objekttypen (*reference types*, Erzeugung auf dem Heap)
- Value types 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)
int j = (int) stack.Pop();   // unboxing
```

- Nachteile:
  - Wrapper-Objekt muss erzeugt werden
  - Man erkennt nicht auf Anhieb, dass das Verfahren teuer ist

B.13 Typunifizierung / Boxing (2)

- Boxing bzw. Unboxing auch bei Strukturen
  - Strukturen besitzen alle Methoden, welche die **`object`**-Klasse besitzt
- 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

B.14 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
if (c == d) { ... }                          // false
if ((object)a == (object)b) { ... }          // false
if (((IComparable)c).CompareTo(d) == 0) {...}  // true
```


B.15 Polymorphie und dynamisches Binden (1)

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

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

- **`override`**-Keyword zum Überschreiben virtueller Methoden

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

- **`new`**-Keyword zum Überschreiben nicht-virtueller Methoden

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


B.15 Polymorphie und dynamisches Binden (2)

- Beispiel

```
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!)
```


B.16 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();
```

B.17 Aufrufsemantik (1)

- Standardmäßig wird “call by value” verwendet
- “call by reference” durch Schlüsselwort **`ref`** oder **`out`**
- **`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 = ...;
    }
}
```


B.17 Aufrufsemantik (2)

- **`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 oder eines Arrays notwendig)

B.18 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;
    }
}
```


B.19 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;
    }
}
```

B.20 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");// Alternative 3: Type.GetType()
// Format: <namespace>.<classname>, <assemblyname>
```

- 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.21 Attribute (1)

- zusätzliche Informationen (z.B. Zugriffsbeschränkungen)

```
[AuthorAttribute ("Bill Gates")]
class Windows
{
    [Localizable(true)]
    public String Text {
        get { return text; }
        ...
```

  - bleibt im MSIL erhalten, kann zur Laufzeit ausgelesen werden
  - ähnlich "**/** */**" oder "**@tag**" in Java, die allerdings nicht in den Bytecode übernommen werden
- **[ConditionalAttribute]**
  - bedinge Kompilierung; entspricht **#ifdef** von C/C++
- **[Obsolete("this class is obolete", false)]**
  - Compilerwarnung oder -fehler
- Attribute müssen als Klassen definiert werden

B.21 Attribute (2)

- Beispiel: Aspektorientierte Programmierung mit Attributen

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

class Window : IWindow {
    [Trace (Level = 1)]
    public void Show() { ... }
}

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

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


B.22 Attribute (3)

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

```
public WindowProxy() {
    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.23Strings

- Objekte der Klasse **`string`** sind konstante, invariante Objekte
  - bei Modifikationen wird neues **`string`**-Objekt als Rückgabewert geliefert
- Vergleich von **`string`**-Objekten

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

if (s1 == s2) { ... }               // true (1)
if ((Object)s1 == (Object)s2) {...} // false (2)
if (s1.Equals(s2)) { ... }          // true (3)
```

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

B.24Zeigerarithmetik

- Zeiger können in C# verwendet werden
- Methoden, die Zeiger enthalten, müssen mit **`unsafe`** markiert werden
- Diese Kennzeichnung soll den Gebrauch von Zeigern minimieren

B.25Konstruktoren & 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
- 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.26Garbage 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)