

DEPARTMENT WIRTSCHAFTSINFORMATIK

FACHBEREICH WIRTSCHAFTSWISSENSCHAFT

Programmieren für Wirtschaftswissenschaftler

SS 2015

Lucian Ionescu

Blockveranstaltung 16.03–27.3.2015

6. Klassen

Agenda

- **Klassen und Objekte**
 - Motivation
 - Grundbegriffe
 - Erstellung
- Datenfelder in Objekten
- Funktionen in Objekten

Wo stehen wir?

- Bisher behandelte Themen
 - das erste C#-Programm (Ein- und Ausgabe)
 - Variablen, Datentypen und Operatoren
 - Verzweigungen und Schleifen
 - Arrays und andere Aufzählungstypen
 - Methoden
- Nächstes (und abschließendes) Thema
 - Klassen und Objekte



Inhalte

Grundlagen

Erstes Programm



Ein- und Ausgabe

Objektorientierung

Variablen



Wie speichert man
Werte zur späteren
Verwendung ab?

Operatoren



Wie verarbeitet und
verknüpft man
Werte?

Verzweigungen und
Schleifen



Wie bildet man
Bedingungen und
Wiederholungen ab?

Erweiterungen

Datenstrukturen



Wie speichert man
gleichartige Elemente
strukturiert ab?

Methoden und
Funktionen



Wie unterteilt man
ein Programm in
Arbeitsschritte?

Später: Wie bildet
man das Verhalten
von Objekten ab?

Klassen

Wie kann man die
reale Situation
modellieren?

Wie bildet man
Objekte und deren
Interaktionen ab?

Grafische
Oberflächen

Wie erstellt man eine
grafische Oberfläche
und Fenster?

Warum gibt es Klassen?

- Verbinden von Daten und Funktionen zu einer strukturellen Einheit
 - „Eine Person **besitzt** einen Namen, eine Adresse und **fährt** ein bestimmtes Auto“
- Stärkere Validierung der Daten
 - „Das Alter einer Person darf nicht negativ sein“
 - „Die Note einer Klausur muss zwischen 1,0 und 5,0 liegen“
 - Wer sollte sich um diese Prüfungen kümmern?
- Leichte Wiederverwendbarkeit
 - Klassen können wiederverwendet werden
 - Mehrere Kunden-Objekte für eine Firma erstellen
 - Aber auch Kunden-Objekte in einem anderen Kontext
 - Definition von Schnittstellen, die nur gültige Daten zulassen
 - Ein Objekt ist für die Konsistenz seiner eigenen Daten selbst verantwortlich
 - Konsistenzprüfungen werden also intern gemacht und müssen nicht vom Anwender/Ersteller des Objekts gemacht werden

Analogie zur realen Welt

- Abbildung der realen Welt durch ein **abstrahiertes Modell**
 - nicht alle Informationen und Prozesse sind für eine Problemstellung relevant
 - Objekte haben **Eigenschaften** und **Funktionen** und agieren miteinander
- Beispiel: (*abstrakte*) Klasse Mensch
 - mögliche **Attribute**: Name, Alter, Geschlecht, ...
 - Instanzen dieser Klasse sind einzelne **Objekte**, also „Pierre-Emerick“ und „Marco“
 - die Attribute können **unterschiedliche Ausprägungen** haben (Name, Alter, ...)
- Eine Klasse dient also als **Bauplan** für Objekte
 - gibt die **Struktur** vor, aber nicht die konkreten **Werte**
 - Erzeugen von Objekten und Zuweisung der Ausprägungen/Werte geschieht erst während des Programm-Ablaufs (und sind nach Ablauf des Programms wieder verschwunden)

Programmier-Paradigmen

Programmiersprachen lassen sich anhand gemeinsamer Eigenschaften verschiedenen **Paradigmen** zuordnen, z.B.:

- **Imperative Programmierung**
 - Programm als Folge einfacher Anweisungen
- **Strukturierte Programmierung**
 - Programmfluss wird durch Strukturen gesteuert
 - keine „wilden Sprünge“ („gehe zurück zu Zeile 2“)
- **Prozedurale Programmierung**
 - Anweisungen werden in Unterprogrammen/Prozeduren gruppiert
- **Modulare Programmierung**
 - Anweisungen werden in Modulen zusammengefasst
- ***Objektorientierte Programmierung***
 - *Gruppierung von Anweisungen und Daten*

Drei Säulen der Objektorientierung

Vererbung

- **Hierarchische Anordnung** von Klassen
- Klassen können von anderen (allgemeineren) Klassen abgeleitet werden
- z.B. Mensch von Lebewesen (**ist-Beziehung**)

Kapselung

- Abschotten der internen Implementierung vor direktem Zugriff
- **Schnittstellen** zur Kommunikation nach außen
- so werden Programmteile austauschbar **und unabhängig voneinander**

Polymorphie

- Objekte können **Objekte anderer Klassen referenzieren**, ohne dass die genaue Ausprägung des referenzierten Objekts bekannt sein muss
- es gilt ein **Versprechen**, dass das referenzierte Objekt gewisse Funktionen und Daten hat

Grundbegriffe der Objektorientierung

- Eine Klasse ist eine **vordefinierte Struktur**
- Objekt
 - **Instanzen** einer Klasse werden **Objekte** genannt
 - von einer Klasse können beliebig viele Objekte erzeugt werden
- Member(s)
 - Elemente einer Klasse werden Member genannt
 - es gibt **funktionale Member (Objekt-Methoden)** und **Daten-Member (Attribute)**
- Konstruktor-Methode
 - ein Konstruktor ist eine spezielle Methode einer Klasse, mit der ein Objekt initialisiert wird
 - Wird benötigt, da die Erstellung von Objekten aus Klassen komplexer ist als z.B. einfache Datentypen wie Zahlen
 - es können mehrere Konstruktoren für eine Klasse existieren (vgl. Überladung von Methoden), das ist in unserem Kontext aber noch nicht relevant

Wie erstellt man eine Klasse

- Allgemeine Syntax

```
class <Identifizier>
{
    <expression>
}
```

- Innerhalb des **Klassenrumpfes** können die Member definiert werden
 - Daten-Member wie Variablen
 - Funktionale Member wie Methoden, Konstruktoren etc.
- Objekt einer Klasse erzeugen
 - Allgemeine Syntax: `<class> <objectname> = new <class>();`
 - Beispiel: `Random rnd = new Random();`

Agenda

- Klassen und Objekte
 - Motivation
 - Grundbegriffe
 - Erstellung
- **Datenfelder in Objekten**
- Funktionen in Objekten

Innerhalb der Klasse (d.h. zwischen den geschweiften Klammern) können in beliebiger Reihenfolge Variablen (Datenfelder) oder Funktionen deklariert werden.

Der Übersicht wegen bietet es sich aber an, zunächst die Datenfelder zu deklarieren und dann die Funktionen folgen zu lassen.

```
class EineKlasse
{
    //Datenfelder...
    public string ersteVariable;
    private bool nochEineVariable;
    private List<AndereKlasse> oderAuchEineListe;

    //...dann der Konstruktor
    public EineKlasse(...)
    {
        //...
    }

    //...und dann die Funktionen
    public void TueEtwas()
    {
        //...
    }
}
```

Objektvariablen

- Klassen können Variablen besitzen, die jeweils an **ein Objekt** gebunden sind
- Diese Variablen werden **Objektvariablen** genannt
- Beispiel:
 - Bello und Ida sind zwei Hunde
 - Dennoch haben die beiden Objekte bis auf ihre gleichartige Struktur nichts miteinander zu tun
 - Jeder Hund kann laufen, aber nur, wenn er auch (in Form eines Objektes) existiert!
 - Wenn Bello läuft, läuft Ida nicht zwangsläufig auch -> es ist eine Objektfunktion!

Beim ersten Punkt des Beispiels muss man ein wenig aufpassen. Um die Analogie zur Umgangssprache mal zu verdeutlichen:

Bello und Ida sind an sich noch keine Hunde, sondern (jetzt in Programmiersprache gesprochen) Bello und Ida sind die Einträge in der Objektvariable „Name“ von zwei Hund-Objekten.

Objektvariablen (2)

- Allgemeine Syntax

```
class <ClassIdentifier>
{
    <modifier1> <datatype1> <identifier1>;
    <modifier2> <datatype2> <identifier2>;
    ...
}
```

- **Modifier** regulieren den Zugriff auf die Member einer Klasse von außen

Modifier	Beschreibung
public	Auf die Variable kann von außerhalb der Klasse zugegriffen werden
private	Zugriff nur durch Methoden innerhalb der Klasse

Man kann Modifier auch weglassen, aber schreibt sie besser immer hin. Gerade am Anfang der Entwicklung kann man erstmal alles auf public setzen. Mit der Zeit kommt dann schnell ein Verständnis, was vielleicht private sein sollte.

Objektvariablen (3)

- Beispiel

```
class Human
{
    public string name;           // Name
    private bool married;        // Verheiratet?
    private List<Human> friends;  // Liste von Human-Objekten

    //und dann noch funktionale Member...
}
```

- Nach Erstellung eines **Objekts** der Klasse `Human` kann der Name beliebig ausgelesen oder verändert werden
- Das Feld `married` ist nur innerhalb des Objekts sichtbar und kann von außen nicht verändert werden
- Dadurch vermeidet man ein willkürliches (gar nicht mal zwingend böswilliges) Manipulation der Variablen
- Man kann eine Variable so „read-only“ machen, dazu gleich mehr bei Objektfunktionen!

Agenda

- Klassen und Objekte
 - Motivation
 - Grundbegriffe
 - Erstellung
- Datenfelder in Objekten
- **Funktionen in Objekten**

Funktionale Member

- Objekte können neben Daten auch **Funktionen** haben
- Beispiel:
 - ein Objekt der Klasse `Human` soll ein anderes Objekt der Klasse `Human` heiraten können, aber nur sofern es sich in der Liste seiner Freunde befindet
 - Es gibt also Regeln die zu beachten sind, und dafür braucht man eine Funktion!
 - Um die Einhaltung der Regeln kümmert sich die die Funktion selbst, das Objekt achtet also selbst auf seine Konsistenz

```
public bool Marry (Human person)
{
    if(!married && friends.Contains(person)) // Reihenfolge der Prüfung!
        married = true;
    else married = false;

    return married;
}
```

Wenn jemand schon verheiratet ist und nochmal heiraten möchte, wird man durch diese Funktion automatisch geschieden. Macht irgendwo sicher Sinn, wenn es nicht beabsichtigt ist, sollte man die Methode aber besser anpassen.

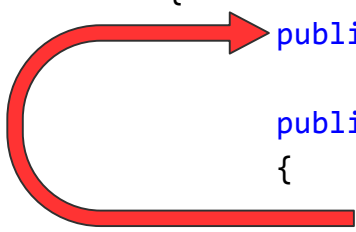
- Kleine Denkaufgabe: Was ist (logisch) möglicherweise nicht korrekt?

Spezialfall: Konstruktoren

- Konstruktoren sind spezielle Methoden ohne Rückgabewert, die **zum Erzeugen eines Objektes** aufgerufen werden
- Hauptfunktion: **Initialisierung** der Daten-Member und Konsistenzprüfung der übergebenen Werte
- Konstruktoren haben immer **denselben Namen** wie die Klasse

```
class Human
{
    public string name;

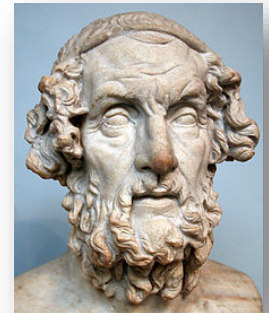
    public Human(string name)
    {
        this.name = name; // Was bewirkt das this?
    }
}
```



Das this haben wir gestern ja schon kurz angesprochen. Mit einem this. vor einer Variable mache ich klar, dass ich die aus *diesem* Objekt meine. Lässt man es hier weg, weißt man der Variable name ihren eigenen Wert zu, was ja nicht beabsichtigt ist. Alternative: einfach unterschiedliche Variablenamen nutzen!

- Beispiel: **Aufruf** eines Konstruktors

```
static void Main(string[] args)
{
    Human homer = new Human("Homer");
}
```



Konstruktoren (2)

- Was passiert, wenn der *Modifier* eines Konstruktors nicht **public** definiert ist?

Dann lässt sich der Konstruktor nicht „von außen“ aufrufen, z.B. aus der Main-Methode.

- Worin müssen sich Konstruktoren unterscheiden, wenn mehrere davon in einer Klasse existieren?

Der Name muss ja dem der Klasse entsprechen. Aber die Parameterliste kann unterschiedlich sein. (Wenn die sich auch gleichen würde, sollte man sich eh fragen, ob man dann mehrere Konstruktoren braucht...)

- Was passiert, wenn kein Konstruktor erstellt wird?
 - dann existiert implizit ein **Standardkonstruktor** ohne Parameter
 - keine initiale Zuweisung von Daten in dem erstellten Objekt, alle Datenfelder bleiben uninitialisiert
 - daher möglichst vermeiden
 - wird ein Konstruktor definiert, so ist dieser implizite Standardkonstruktor nicht mehr verfügbar

Zusätzliche Initialisierung

- Nicht alles muss über den Konstruktor initialisiert werden, wenn Standardwerte bekannt sind
- Beispiel
 - man kann die Freundesliste **entweder** gleich mit der Deklaration der Objektvariablen initialisieren... (A)
 - ...**oder** im Konstruktor initialisieren (B)

```
class Human
{
    private List<Human> friends = new List<Human>(); // (A)
    public string name;

    public Human(string name)
    {
        this.name = name;
        friends = new List<Human>(); // (B)
    }
}
```

- (A) macht insbesondere bei komplexen Datenstrukturen (Listen etc.) Sinn, der Variablen *friends* wird hier also eine leere Liste zugewiesen, die dann später genutzt werden kann

Die null-Referenz

- Manchmal möchte man ein Objekt nicht direkt erstellen, sondern z.B. in einer Verzweigung unterschiedlich erzeugen
- Dann kann man das Schlüsselwort `null` nutzen
- Ein Verweis auf `null` ist ein Verweis „ins Leere“, d.h. auf kein Objekt
- Alle nicht initialisierten Verweise zeigen auf `null`!
- Nutzung kann auch zu Fehlern führen, wenn ein Objekt erwartet wird
- In einem solchen Fall also abfragen (einfacher Vergleichsoperator)
- Beispiel:

```
Human newHuman = null; // zeigt nach Deklaration auf null
if (!earthFull)
    newHuman = new Human("Homer");
if (newHuman != null) { // wenn newHuman nicht auf null zeigt
    InitializeGeburtsParty();
}
```

Vielen Dank für die „Aufmerksamkeit“!

