
Continental
The Future in Motion
C1TT
C#
Objecktorientierte Programmierung
C1TT
Continental Institut
für Technologie
und Transformation

KLASSEN

Klassen

- Funktionalität wie Strukturen
- Lassen sich von anderen Klassen ableiten
  - Erben Funktionalität d. Basisklasse
  - Fördern Organisation + Wiederverwendbarkeit d. Codes
  - Eine Klasse kann immer nur **eine** Klasse zur gleichen Zeit erben
- Sind Referenztypen
  - Parameterübergabe speichereffizienter
- `class`-Schlüsselwort
- Optional Basisklassenangabe und `implements`
- Nicht automatisch instanziiert, **`new`** benutzen!

Generische Typen

- Vorteile generischer Typen
  - Typsicher
  - Keine Typumwandlungen
  - Kein Boxing
- Mögliche generische Typen können eingeschränkt werden durch
  - where T : <IMyInterface>
  - where T : <MyBaseClass>
  - where T : U
  - where T : new()
  - where T : struct
  - where T : class

Ziele des objektorientierten Programmierens

Continental Institut für Technologie und Transformation

- Objektorientierung
  - Ermöglicht, reale Objekte und Vorgänge im Programm abzubilden
  - Programmierer kann auf niedrigerer Abstraktionsstufe arbeiten
  - Resultat kontinuierlicher Entwicklung
    - Einfachster Datentyp: Binärsystem
    - Praktischere Datentypen: String, Integer
    - Noch praktischere Datentypen: Strukturen, z. B. *Adresse*
    - Datentypen entsprechen Realität: Objekte mit Verhalten

Klassen und Objekte

- Klasse
  - Bildet mit Eigenschaften, Methoden + Ereignissen einen Gegenstand, Prozess oder Rolle aus realer Welt nach
  - Auch.NET Framework ordnet seine Funktionalität in Klassen
  - Bauplan, nach dem sich beliebig viele Objekte herstellen lassen
- Objekt
  - Zustand
  - Instanz, Instanziieren
- Instanziieren
  - Klassen sind Referenztypen
    - Müssen im Gegensatz zu Strukturen explizit instanziiert werden
    - Schlüsselwort new: `Object o = new Object();`
  - `NullReferenceException`!
- Generische, partielle und statische Klassen

Klassen ableiten

- Neue Klasse
  - Ableiten von einer Basisklasse
    - Basisklasse darf **nicht** als `sealed` deklariert sein
  - Erbt alle Eigenschaften, Methoden und Ereignisse der Basisklasse
  - Erbt keine Konstruktoren
  - Implementierungsvererbung
  - Zusätzliche Mitglieder implementieren
  - Funktionalität geerbter Mitglieder durch überschreiben verändern
- Keine Mehrfachvererbung, aber beliebig viele Schnittstellen implementierbar
- Instanz
  - Schlüsselwort `this` -> Instanz Mitglieder
  - Schlüsselwort `base` -> Instanz Mitglieder der Basisklasse
- Destruktoren möglich (Garbage Collector)

Abstrakte Klassen

- Abstrakte Klasse
  - Liefert nur Grundriss für weitere Klassen
  - Kann selber gar nicht instanziiert werden
  - Methode ohne Implementierung -> abstrakte Methode
  - Klasse mit min. einer abstrakten Methode -> abstrakte Klasse
  - Darf nicht instanziiert werden, Versuch -> Fehlermeldung des Compilers

Eigenschaften + Felder (1/2)

- Eigenschaft
  - Enthält Werte einer Instanz
  - Implementieren am einfachsten durch öffentliches Feld
  - Stattdessen meistens privates Feld + spezielle Methoden
    - Zugriff kontrollierbar, z. B. beschränken auf Wertebereiche
    - Schreibschutz möglich
    - Nebeneffekte auslösen, z. B. konvertieren oder berechnen
    - Zugriff im Code sieht aus wie Zugriff auf öffentliches Feld
    - Zugriffsmodifizierer

- Eigenschaftendeklaration
  - Privates / geschütztes Feld
  - `get`- + `set`-Methode / Accessoren / Accessor + Mutator
  - `set`-Accessor
    - Neuer Wert = `value`
    - Weglassen = schreibgeschützte Eigenschaft

Eigenschaften + Felder (2/2)

- Schreibgeschützte Eigenschaft auch durch readonly-Feld
- Indexer
  - Ermöglicht Zugriff über Array-Syntax
  - 1 pro Klasse
  - Namenlose Eigenschaft
- Asymmetrische Accessoren

Methoden (1/3)

- Methode
  - Definiert Verhalten einer Klasse oder Struktur
  - Beliebige Anzahl von Parametern + 1 Rückgabewert (void falls nichts zurückgegeben wird)
  - Überladen
    - Mehrere Methoden gleichen Namens
    - Typ oder Anzahl der Parameter unterschiedlich
    - Compiler setzt automatisch entsprechend Parametern richtige Methode ein
    - Ersetzt Methoden mit Default-Parametern
    - C# unterstützt ab Version 4.0 auch optionale und benannte Parameter
  - generische + anonyme Methoden
  - Extension Methods

Methoden (2/3)

- Parameterübergabe
  - Werttyp
    - Kopie des Originalwerts
    - Veränderung des Parameterwerts beeinflusst nicht den Originalwert
  - Referenztyp
    - Kopie des Verweises
    - Beide Verweise referenzieren dasselbe Objekt!
    - Änderung in Methode = Änderung des Originalobjekts
  - Mit `ref` und `out` Übergabeverhalten einstellbar
  - Mit `params` Array-Parameter bzw. offene Parameterlisten definieren

Methoden (3/3)

- Vererbung
  - Zugriffsmodifizierer
  - Überschreiben
    - `virtual` in Basisklasse
    - `override` in abgeleiteter Klasse
    - `new` Neu-Implementieren bei Namensgleichheit
    - `abstract` Keine Standard-Implementierung, Überschreiben erforderlich

Konstruktoren

- Konstruktor
  - Versetzt Instanz einer Klasse in definierten Zustand
  - Vorteile gegenüber Methoden + Eigenschaften
  - Beim Instanziieren einer Klasse automatisch aufgerufen
  - Innerhalb der Klasse nur aus anderem Konstruktor aufrufbar
  - Initialisierer
    - `this` -> anderer Konstruktor der Klasse
    - `base` -> Konstruktor der Basisklasse
  - Parameterloser Standard Konstruktor
    - Im Quelltext normalerweise nicht sichtbar
    - Wird versteckt, sobald Klasse zusätzliche Konstruktoren implementiert
    - Kann explizit wieder implementiert werden
  - Beliebig viele Konstruktoren mit unterschiedlicher Signatur möglich

Statische Mitglieder

- Statische Mitglieder
  - Ohne Instanz ihrer Klasse bzw. Struktur verfügbar
  - z. B. `System.String` Klasse
    - Statische Eigenschaft `Empty`
    - Statische Methoden wie `Compare()`, `Concat()` und `Copy()`
  - Auch Felder + Ereignisse + Konstruktoren können statisch sein
  - Schlüsselwort `static`
  - Können nicht auf Instanz-Mitglieder zugreifen, kein `this` oder `base`
  - Zugriff nur über Typ, **nicht** über Instanz
  - Alle Instanzen des Typs teilen sich statische Mitglieder -> Instanz-Zähler
  - Statische Klassen

ARBEITEN MIT LOKALEN DATEIEN UND VERZEICHNISSEN

Die `File` und `FileInfo` Klassen

- Namespace `System.IO`
- Die `File` Klasse bietet atomare Methoden zum Lesen und Schreiben von Dateien an:
  - Zum Lesen:
    - `ReadAllText()`
    - `ReadAllLines()`
    - `ReadAllBytes()`
  - Zum Schreiben entsprechend:
    - `WriteAllXXX()`
    - `AppendAllXXX()`
- Die `File` Klasse bietet statische Elemente an wie
  - `File.Delete()`
  - `File.Exists()`
- Die `FileInfo` Klasse bietet Instanz Elemente an wie
  - `myFile.Delete()`
  - `myFile.Directory()`
  - `myFile.Exists`

Die `Directory` und `DirectoryInfo` Klassen

- `Directory` mit statischen Elementen
  - `Directory.Delete()`
  - `Directory.Exists()`
  - `Directory.GetFiles()`
- DirectoryInfo mit Instanz Elementen
  - `myDirectory.FullName`
  - `myDirectory.Exists`
  - `myDirectory.GetFiles()`

Die `Path` Klasse

- Kapselt viele I/O Funktionen
- Vorteile:
  - Weniger Code
  - Zeitersparnis
  - Konzentration auf komplexere I/O Funktionen
- Bietet u.a. folgende statischen Methoden an
  - `Path.HasExtension()`
  - `Path.GetExtension()`
  - `Path.GetTempFileName()`
  - `Path.GetTempPath()`

DATEN MIT LINQ ABFRAGEN

LINQ Einführung

- LINQ = **L**anguage **In**tegrated **Q**uery
- Namespace `System.Linq`
- LINQ ist
  - Standardisiert
    → Die Syntax bleibt gleich, egal welche Datenquelle man abfragt
  - Deklarativ
    → Programmierkonzept, das beschreibt was man tut möchte, ohne beschreiben zu müssen wie es getan wird
- Ähnelt klassischer SQL Syntax

LINQ verwenden

- LINQ kann entweder als Ausdruck verwendet werden:

```
var ergebnis = from element in sammlung
               orderby element.Eigenschaft1 ascending
               select element;
```

- Oder als Erweiterungsmethode:

```
var erstesElement = sammlung.FirstOrDefault();
var anzahl = sammlung.Count();
```

SERIALISIERUNG

Unterstützte Formate

- .NET unterstützt 3 unterschiedliche Formate der Serialisierung
  - Binär

```
10101010101011111010110101010110101111111010101101100001
```

  - XML

```
<SOAP-ENV:Body>
        <a1:MeinSerialisierbarerTyp id="ref-1"
                xmlns:a1="NAMESPACE+ASSEMBLYNAME+VERSION">
                <SomeInt>1</SomeInt>
                <SomeString id="ref-3">Irgendein Text</SomeString>
        </a1:MeinSerialisierbarerTyp>
</SOAP-ENV:Body>
```

  - JSON

```
{
   "Eigenschaft1":"Wert1",
   "Eigenschaft2":"Wert2"
}
```

Beispiel eines eigenen, serialisierbaren Typs


```
[Serializable]
public class MeinSerialisierbarerTyp : ISerializable
{
    public MeinSerialisierbarerTyp()
    { }

    protected MeinSerialisierbarerTyp(SerializationInfo info,
                                      StreamingContext context)
    {}

    public virtual void GetObjectData(SerializationInfo info,
                                      StreamingContext context)
    {}
}
```

Die "Formatierer"

- `BinaryFormatter`
- `SoapFormatter`
- `DataContractJSONSerializer`
- `IFormatter`
- `ISerializable.Serialize()`

- Ablauf der `Serialize()` Methode
  1. Formatter prüft, ob Surrogate Selector existiert
     1. Wenn ja, Prüfung, ob dieser Objekte des übergebenen Typs behandelt und Ausführung von `ISerializable.GetObjectData()`
     2. Wenn nein oder der Typ nicht behandelt wird → Suche nach `Serializable` Attribut
        1. Existiert dieses Attribut nicht ➔ `SerializationException`
        2. Existiert das Attribut, erneut Prüfung auf `ISerializable` → Ausführung von `GetObjectData()`
        3. Existiert das Attribut, `ISerializable` aber nicht, wird Standard-Serialisierung für **alle** Eigenschaften genutzt, die **nicht** als `NonSerialized` gekennzeichnet wurden