SANDATA | IT-Trainingszentrum GmbH
Die IT-Gruppe

C#-Seminar

Dr.sc.nat. Michael J.M. Wagner*

Revision 330-2022

*michael@wagnertech.de

Inhaltsverzeichnis

1 Einführung **1**

1.1 C# und .NET . . . 1

1.2 Visual Studio . . . 2

2 Sprachgrundlagen **2**

2.1 Anwendungen erstellen . . . 2

2.2 Sprachelemente . . . 3

2.3 Kontrollstrukturen . . . 7

2.4 Prozeduraufruf . . . 8

3 System-, Datei- und Laufwerkszugriffe **10**

4 Verarbeitung von Zeichenketten **12**

5 Objektorientierung **12**

5.1 Klassen, Felder und Methoden . . . 12

5.2 Kapselung und Konstruktoren . . . 15

5.3 Vererbung . . . 18

5.4 Polymorphismus . . . 19

5.5 Abstrakte Klassen und Methoden . . . 19

5.6 Schnittstellen . . . 20

6 Komplexe Datentypen **21**

6.1 Eindimensionale Arrays . . . 21

6.2 Mehrdimensionale Arrays . . . 22

6.3 Auflistungen . . . 22

6.4 Aufzählungstypen . . . 24

6.5 Generische Typen . . . 25

7 Fehlerbehandlung **26**

8 Spezielle Themen **27**

8.1 XML-Serialisierung . . . 27

8.2 Delegaten und Ereignisse . . . 28

9 Quellen **30**

1 Einführung

1.1 C# und .NET[1]

C#

- C# ist eine objektorientierte Programmiersprache von Microsoft
- „Das bessere Java“
- C# läuft auf .NET.

Visual Studio: IDE von Microsoft für

- Visual Basic
- C#
- C++
- ...

.NET

> .NET ... beschreibt eine Softwareumgebung zur Entwicklung *programmiersprachen- und plattformunabhängiger* Software, in dem Programmcode verschiedener Programmiersprachen mittels eines Compilers in eine Zwischensprache übersetzt wird, die für alle Programmiersprachen gleich ist. Diese Zwischensprache ist plattformneutral und kann auf unterschiedlichen Betriebssystemen wie Windows oder Linux ausgeführt werden.[2]

.NET besteht aus

- umfangreicher Klassenbibliothek, programmierspachenunabhängig
- *common language runtime* (CLR)
- *just in time compiler* (JIT)

Assembly

- Produkt des Compilers (exe oder dll)
- *Microsoft intermediate language* (MSIL)
- benötigt CLR um ausgeführt werden zu können
- enthält Meta-Information

Bei der Ausführung eines Assemply übersetzt der JIT der .NET-Plattform die MSIL in Maschinencode (S. Abb. 1).

[1] VC2015:Kap. 2
[2] VC2012: S. 7
[3] VC2012: S. 7.

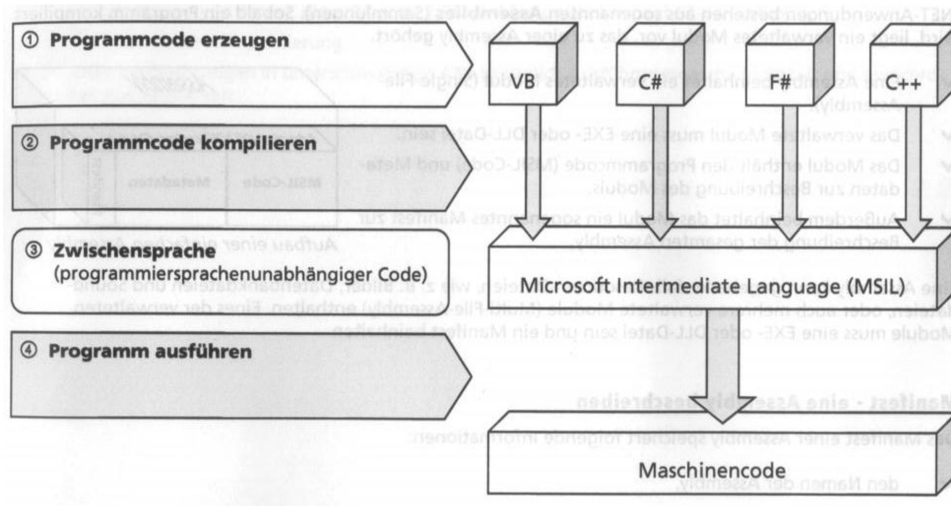


Abbildung 1: Programme gemäß .NET entwickeln[3]

1.2 Visual Studio

Visual Studio

- Microsoft IDE
- Weitere IDEs:
  - Eclipse: Für viele Programmiersprachen, Java-basiert
  - VSCode: Microsoft, offen für Programmiersprachen, Plattformen
- in der Bedienung ähnlich

Strukturierung des Codes

- Projektmappe/Solution, Workspace
- Projekt → Programm oder Bibliothek

Strukturierung der IDE

- Die Fensteranordnung kann individuell angepasst und abgespeichert werden.

2 Sprachgrundlagen

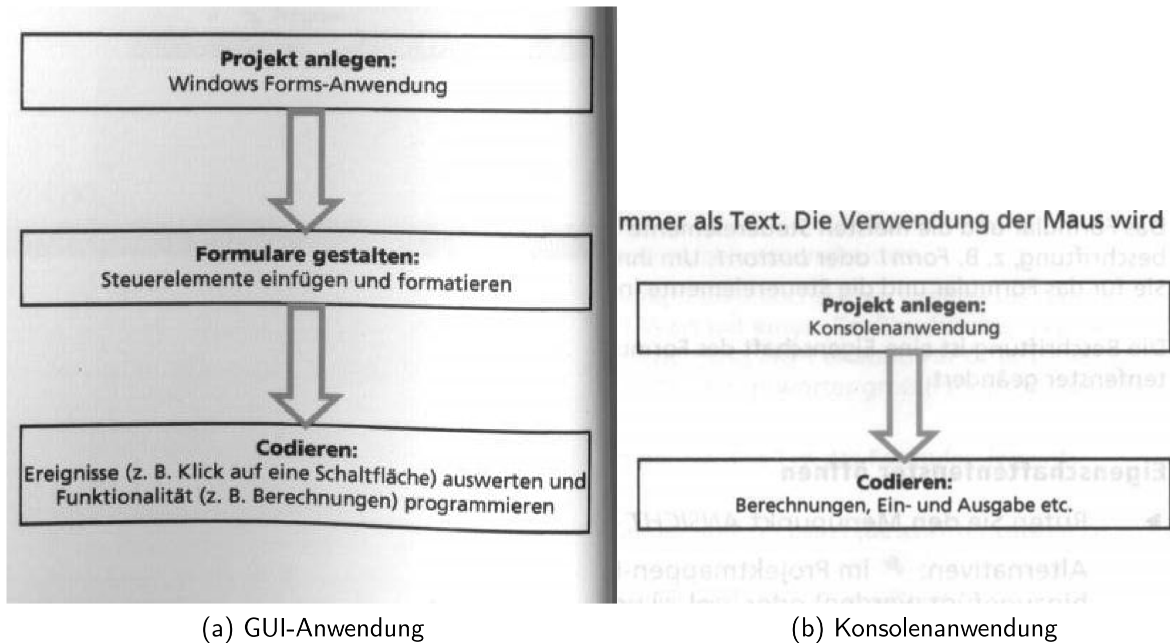
2.1 Anwendungen erstellen[4]

Die einfachsten Formen einer Anwendung sind:

- WindowsForms-Anwendung (Windows)
- GTK-Anwendung (Linux)

[4] VC2015: Kap. 4

- Konsolen-Anwendung



(a) GUI-Anwendung (b) Konsolenanwendung

Abbildung 2: Anwendungserstellung

Konsolenanwendung

Konsolenanwendungen sind nützlich für:

- Alternativer Zugang zum Anwendungskern
- Testskripte
- Automatisierung

Aufgabe:

Erstellen Sie mit Ihrer Softwareentwicklungsumgebung eine "Hallo Welt" - Konsolenanwendung.

2.2 Sprachelemente

Bezeichner und Schlüsselwörter[5]

- Regeln wie in C/C++/Java
- Schlüsselwörter [VC:6.2]

[5] VC2015: Kap. 6.2

Aufbau eines Programms[6]

Jede Anwendung benötigt ein Hauptprogramm: Irgendeine Klasse mit einer `Main(string[] args)` – Methode.

Programmcode dokumentieren[7]

`//` : Zeilenkommentar

`/* ... */` : Blockkommentar

`///` : Dokumentation, die extrahiert werden kann

Anweisungen in Visual C# erstellen

Anweisungen in Visual C# erstellen[8]

- Anweisungen werden mit ";" abgeschlossen
- Anweisungen werden mit `{ ... }` zu Blöcken zusammengefasst

Einfache Datentypen[9]

(1) Numerische Datentypen

Integer-Datentypen [VC:6.6]

Gleitkomma-Datentypen [VC:6.6a]

(2) Zeichenketten-Datentypen [VC:6.6b]

(3) Boolescher (logischer) Datentyp [VC:6.6c]

Literale[10]

- Zahlen
- Escape-Sequenzen werden in Zeichenketten mit "\" eingeleitet.
- Für Windows-Pfade gibt es Strings ohne „Escape": `@"c:\Users\Kurs"`

[6] VC2015: Kap. 6.3
[7] VC2015: Kap. 6.4
[8] VC2015: Kap. 6.5
[9] VC2015: Kap. 6.6]
[10] VC2015: Kap. 6.7

Mit Variablen arbeiten[11]

- Sichtbarkeit im Block (wie in C/C++/Java)
- Implizite Typisierung mit `var` [VC:6.8]

Typkompatibilität und Typkonversion[12]

- Implizit: `double d = 2;`
- Explizit mit `Convert` / `ToString()` / `Parse()` → Abb. 3

  Zur Beachtung: `Convert` und `Parse()` sind *prozedurale* Aufrufe, `ToString()` hingegen ist *objektorientiert*. Im ersten Fall steht vor dem Punkt das Modul, das die Prozedur enthält, im zweiten die Variable, die den Wert enthält.

Beispiele für Typkonversionen: *TypeCast.sln*

```
    int number = 4567;
    double size = 123.12;
    char ch = 'K';
    string txt;
①   number = (int)size;
②   size = 149;
③   txt = ch.ToString();
④   txt = number.ToString();
⑤   number = Convert.ToInt32("4567");
⑥   size = Double.Parse("4567,89");
⑦   size = Convert.ToDouble("4567,89");
⑧   ch = Convert.ToChar("k");
```

Abbildung 3: Beispiele für Typumwandlungen[13]

Formatierte Ausgabe: `Console.WriteLine ("{0,5};{1:c2}", s, i);`

Erklärung [VC:6.11].

Konstanten[14]

- Konstanten werden mit `const` gekennzeichnet.

[11]VC2015: Kap. 6.8
[12]VC2015: Kap. 6.11
[13]VC2012: S. 66/68.
[14]VC2015: Kap. 6.12

Arithemtische Operatoren und Vorzeichen- und Verkettungsoperatoren[15]

- Arithmetisch: `+ - * / %`
  - Feldüberläufe lassen sich duch eine entsprechende Compilereinstellung oder das Schlüsselword checked überprüfen.
- Vorzeichenoperator: `-`
- Verkettungsoperator: `+`
- Inkrement/Dekrement: `++ --`

Logische Operatoren[16]

- Vergleichsoperatoren: `== != > < >= <=`
- Verknüpfungsoperatoren: `! & && | || ^`

Zuweisungsoperatoren für eine verkürzte Schreibweise verwenden[17]

`a = a + b;` → `a += b;`

> Aufgabe:
>
> Übung 6.16: Werte ein- und ausgeben
>
> Hinweis: Von der Konsole lesen: `string s = Console.ReadLine();`

Namensräume

Damit Klassen keine systemweit eindeutigen Namen haben müssen, wird der Code mittels Namensräumen strukturiert. Überlicherweise wird der Namensraum wie das Assembly genannt. Mit `using` werden Namensräume in den aktuellen Kontext eingebunden.

- Erstellung eines Namensraums [VC:9.5]
- Verwendung eines Namensraums [VC:9.5a]
- Beispiele: `Console.ReadLine(); System.Math.Max(3,5);`

Anweisung (gekürzt)	Zweck
`if ...`	Es folgen Anweisungen, die abhängig von einer Bedingung einmal oder gar nicht ausgeführt werden sollen.
`if ...` `else ...`	Zwei alternative Anweisungsblöcke stehen zur Auswahl. In Abhängigkeit von einer Bedingung wird der `if`- oder `else`-Zweig mit seiner Anweisung oder seinem Anweisungsblock ausgeführt.

`switch ...`
`case ...:`
`...`
`default:`
	In Abhängigkeit von dem Wert eines Ausdrucks soll einer von mehreren Auswahlblöcken ausgeführt werden. Entspricht keine der `case`-Anweisungen dem Selektor, werden die Anweisungen nach `default:` ausgeführt.

`for ...`	Eine oder mehrere Anweisungen werden in Abhängigkeit von einer Schleifenbedingung ausgeführt.
`while ...`	Anweisungen sollen in Abhängigkeit von einer Bedingung einmal, mehrmals oder gar nicht ausgeführt werden. Die Bedingung wird am Anfang der Schleife geprüft.
`do ...` `while ...`	Anweisungen sollen in Abhängigkeit von einer Bedingung mindestens einmal oder mehrmals abgearbeitet werden. Die Bedingungsprüfung findet am Ende der Schleife statt.
`break`	In Abhängigkeit von einer Bedingung kann ein Anweisungsblock der `while`-, `do-while`- `for`- und `switch`-Anweisung abgebrochen werden. Die Kontrollstruktur wird beendet und die nachfolgende Anweisung ausgeführt.
`continue`	In Abhängigkeit von einer Bedingung kann in einer `while`-, `do-while`- oder `for`-Anweisung zur Auswertung der Bedingung gesprungen werden.

Abbildung 4: Übersicht Kontrollstrukturen[19]

2.3 Kontrollstrukturen[18]

Übersicht S. Abb. 4.

Aufgabe:

Übung 7.13: Testauswertung

Aufgabe:

Mit dieser Aufgabe beginnt die Implementierung einer „Büchereiverwaltung".

- Legen Sie ein Projekt *Bucherei* an.

[15]VC2015: Kap. 6.13
[16]VC2015: Kap. 6.14
[17]VC2015: Kap. 6.15
[18]VC2015: Kap. 7
[19]VC2012: S. 93.

- In der Datei `Bucherei.cs` soll abgefragt werden, welche Entität gepflegt werden soll (*Medium, Nutzer, Ausleihe, [Ende]*)
- Danach soll gefragt werden, welche Operation ausgeführt werden soll (*Anlegen, Lesen, Ändern, Löschen, Alle anzeigen*)
- Implementieren Sie eine Schleife um die Eingabe, die erst verlassen wird, wenn die Eingabe gültig ist.
- Fragen Sie je nach dieser Eingabe nach den weiteren benötigten Daten:
  - Anlegen/Ändern von Medien: *Signatur, Autor, Titel, Typ, Seitenzahl, Spieldauer*
  - Lesen/Löschen von Medien: *Signatur*
  - Anlegen von Nutzern: *Nachname, Vorname, Geburtsdatum*
  - Ändern von Nutzern: *Nutzernummer, Nachname, Vorname, Geburtsdatum*
  - Lesen/Löschen von Nutzern: *Nutzernummer*
  - Bei Ausleihen: *Signatur, Nutzernummer*
- Implementieren Sie Schleifen um die Eingaben, die erst verlassen werden, wenn die benötigten Felder gefüllt sind.
- Implementieren Sie um das Ganze eine weitere Schleife, die erst verlassen wird, wenn bei der Abfrage der Entität *Ende* gewählt wird.

2.4 Prozeduraufruf

Mit C# lässt sich neben dem objektorientierten Paradigma auch prozedural programmieren. Klassen sind dann Programmmodule. Im folgenden Beispiel ist skizziert, wie dies funktioniert:

```
Class MainModul{

  Main(string[] args){
    SubModul.ParameterloseMethode();
    SubModul.MethodeMitParameter("Hallo");
    int i;
    i = MethodeMitRuckgabwert("Hallo");
  }
}

class SubModul{

  internal static void ParameterloseMethode(){
    // hier kommt die Implementierung
  }

  internal static void MethodeMitParameter(string param){
    // hier kommt die Verarbeitung des Parameters
  }
```

```
    internal static int MethodeMitRuckgabwert(string param){
      // hier kommt die Verarbeitung des Parameters
      // und die Berechnung des Ruckgabewerts
      int k = 3;
      return k;
    }
}
```

Üblicherweise wird pro Klasse eine gleichnamige Programmdatei verwendet.

In der dargestellten Weise wird der Parameter als Wert übergeben. Wenn die aufgerufene Prozedur `param` ändert, dat dies keine Auswirkung auf das aufrufende Programm (s. Abb 5).

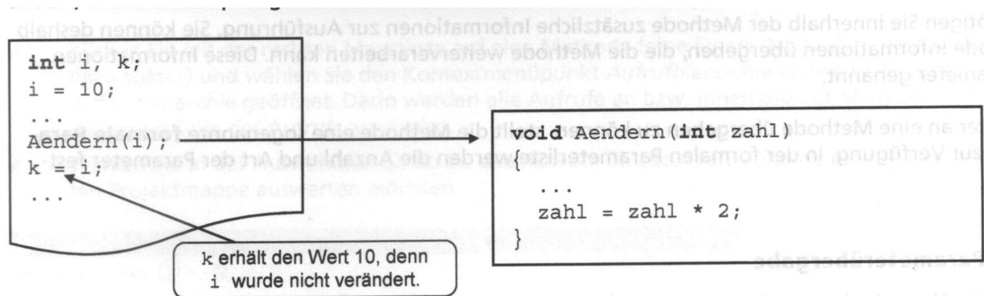


Abbildung 5: Call by Value[20]

Soll eine Rückwirkung auf in das aufrufende Programm erfolgen, so muss der Paramter als Verweis übergeben werden (s. Abb 6).

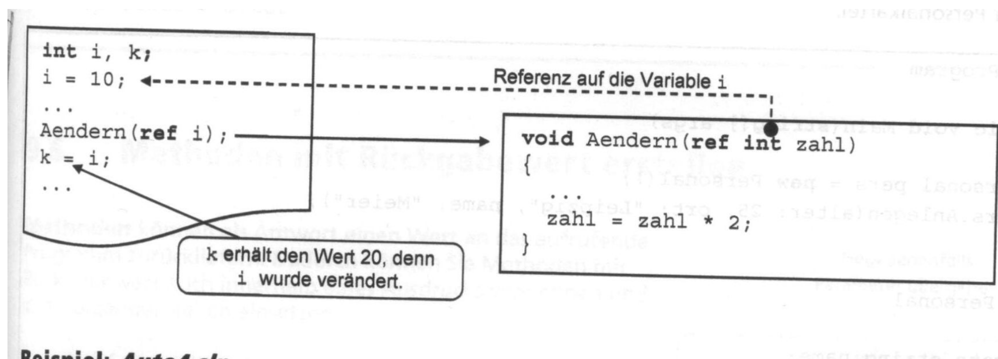


Abbildung 6: Call by Reference[21]

Aufgabe:
Übung 8.12: Parameterübergabe an eine Methode (nur Teil1)

[20]VC2012: S. 106.
[21]VC2012: S. 107.

3 System-, Datei- und Laufwerkszugriffe[22]

Mit Laufwerken, Ordnern und Dateien arbeiten[23]

Übersicht s. Abb. 7.

ie möchten ...	
aufwerksnamen ermitteln	Die Methode `System.IO.DriveInfo.GetDrives()` liefert eine Collection mit den am Computer des Anwenders verfügbaren Laufwerken.
len aktuellen Ordner ermitteln	`System.IO.Directory.GetCurrentDirectory()`
rüfen, ob eine Datei/ein Ordner xistiert	`System.IO.File.Exists()` bzw. `System.IO.Directory.Exists()`
ine Datei kopieren	`System.IO.File.Copy()`
ine Datei/einen Ordner erschieben bzw. umbenennen	`System.IO.File.Move()` bzw. `System.IO.Directory.Move()`
ine Datei/einen Ordner löschen	`System.IO.File.Delete()` bzw. `System.IO.Directory.Delete()`
lie in einem Ordner gespeicheren Dateien ermitteln	`System.IO.Directory.GetFiles()`

Abbildung 7: Befehle zum Arbeiten mit Laufwerken[24]

Mit Textdateien arbeiten[25]

.NET stellt für das Arbeiten mit Textdateien die Klassen `StreamWriter` und `StreamReader` zur Verfügung. Hier der zeilenweise Umgang:

```
using System.IO;

// Zeilenweises Lesen
StreamReader SR;
using (SR = new StreamReader("datei.name")) {
  while (SR.peek() != -1) {
    string line = SR.ReadLine();
    // Verarbeitung der gelesenen Zeile
  }
} // close findet beim Verlassen des Blockes statt

// Zeilenweises Schreiben
StreamWriter SW;
using (SW = new StreamWriter("datei.name")) {
                                  // ...", true) h"angt an Datei an
  for (int i=1; i<10; i++) {
    SW.Write("Das ist Zeile: ");
    SW.WriteLine(i);
```

[22]VC2015: Kap. 15
[23]VC2015: Kap. 15.3
[24]VC2012: S. 218.
[25]VC2015: Kap. 15.4

```
  }
} // close findet beim Verlassen des Blockes statt
```

Weitere Eigenschaften von `StreamWriter` und `StreamReader` → Abb. 8.

Ausgewählte Methoden und Eigenschaften der Klasse `StreamWriter`

Eigenschaft/Methode	Beschreibung
`BaseStream`	Gibt den zugrunde liegenden Stream an.
`Encoding`	Gibt die Codierung der Ausgabe an.
`Close()`	Schließt die `StreamWriter`-Instanz und den zugrunde liegenden Stream. Ein zusätzliches Schließen des Streams ist also nicht unbedingt erforderlich.
`Flush()`	Löscht den Puffer für die aktuelle `StreamWriter`-Instanz und veranlasst zuvor die Ausgabe aller im Puffer befindlichen Daten in den Stream.
`Write()`	Schreibt Daten fortlaufend in den Stream.
`WriteLine()`	Schließt das Schreiben von Daten mit einem Zeilenumbruch ab.

Ausgewählte Methoden und Eigenschaften der Klasse `StreamReader`

Eigenschaft/Methode	Beschreibung
`BaseStream`	Gibt den zu Grunde liegenden Stream an.
`CurrentEncoding`	Ruft die Codierung ab, die von der aktuellen `StreamReader`-Instanz verwendet wird.
`Close()`	Schließt die `StreamReader`-Instanz und den zugrunde liegenden Stream.
`EndOfStream`	Besitzt den Wert `true`, wenn das Ende der Datei erreicht ist.
`Peek()`	Gibt das nächste Zeichen zurück, ohne den Lesezeiger weiterzubewegen. Am Dateiende (kein Zeichen mehr verfügbar) wird -1 zurückgegeben.
`Read()`	Liest das nächste Zeichen oder den nächsten Block von Zeichen aus dem Stream.
`ReadLine()`	Liest eine Zeile von Zeichen aus dem Stream.
`ReadToEnd()`	Liest den Stream bis zum Ende.

Abbildung 8: Eigenschaften der Stream-Klassen[26]

Aufgabe (Bücherei):

- Legen Sie eine Klasse *Nutzerverwaltung* an.
- Schreiben Sie eine Prozedur *Anlegen* mit folgender Signatur:

```
internal static void Anlegen(string nachname, string vorname, string gebdatum)
```

- Diese Prozedur soll die Daten an die Datei `nutzer.csv` kommasepariert anhängen.
- Rufen Sie die Prozedur aus dem Hauptprogramm auf.

[26]VC2012: S. 221.

4 Verarbeitung von Zeichenketten[27]

Bei der Betrachtung der Stringoperationen ist zu beachten, dass diese teilweise dem prozeduralen Paradigma folgen, teilweise dem objektorientierten. Im ersten Fall werden die Methoden in der Form *Modulname.Prozedur(...)* (z.B. `s2 = String.Copy(s);`) aufgerufen, im zweiten Fall in der Form *variablenname.Methode(...)* (z.B. `s2 = s.Remove(1,2);`).

Eine Aufstellung von Methoden zur Stringverarbeitung findet sich in [VC:8.8]. Hier fehlt aber eine weitere nützliche Methode:

Name	Beschreibung	`string s, s2;`

`Split()`	Zerlegt einen String in ein Array von Strings. Das Trennzeichen wird als Parameter übergeben.
	`public string[] Split (char[] separator)`
		`s = "42,12,19";`
		`string[] sa = s.Split(',');`
		`sa[0] => "42"`
		`sa[1] => "12" ...`

Aufgabe (Bücherei):

Implementieren Sie in der `Nutzerverwaltung` die Prozedur `AlleAnzeigen`:

- Prüfen Sie, ob die Datei `nutzer.csv` vorhanden ist. Falls nein, geben Sie „Keine Nutzer im System" aus.
- Lesen Sie die Datei Zeile für Zeile
- Zerlegen Sie jede Zeile in ihre Bestandzeile und geben Sie diese aus.

5 Objektorientierung

5.1 Klassen, Felder und Methoden[28]

Grundlagen der objektorientierten Programmierung

Ausgangspunkt: Strukturierte Programmierung bis in die 80'ger Jahre

- Daten, oft global
- Funktionen und Prozeduren, die auf den Daten arbeiten

Ansatz der Objektorienierung:

- Daten und Funktionen („Methoden“) gehören zusammengefasst
- Zugriff ausschließlich über Schnittstellen (s. Abb. 9)

[27] VC2015: Kap. 8.8
[28] VC2015, Kap. 8

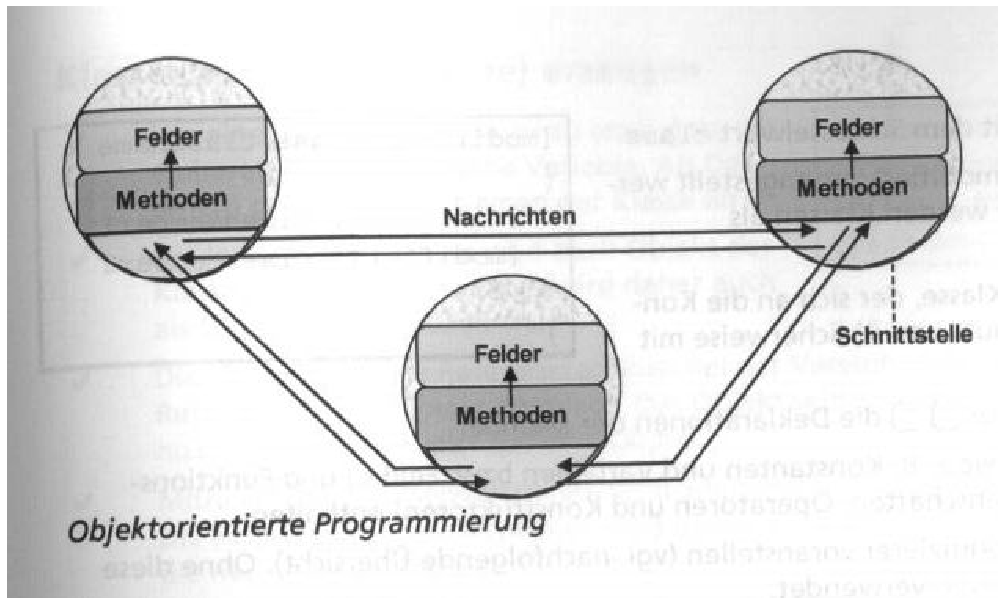


Abbildung 9: Prinzip der Objektorientierung[29]

Klassen und Instanzen

Was ist eine Klasse, was sind Objekte?

> Eine Klasse beschreibt als Bauplan die Gemeinsamkeiten einer Menge von Objekten. Eine Klasse ist somit ein Modell, auf dessen Basis Objekte erstellt werden können. Die Klasse beinhaltet die vollständige Beschreibung dieses Modells. So vereint eine Klasse alle Felder (auch Datenelemente genannt), die diese Klasse kennzeichnen, und alle Methoden zur Verwendung der Daten und zur Beschreibung der Funktionalität. Da die Felder und Methoden zu der Klasse gehören, werden die auch als Member (Mitglieder) der Klasse bezeichnet.
>
> Objekte (Instanzen) stellen konkrete Exemplare der Klasse dar. Auf der Basis einer Klasse können beliebig viele Objekte erzeugt (instanziiert) werden. In den Feldern werden die objektspezifischen Werte für das jeweilige Objekt gespeichert, während die Methoden Aktionen ausführen. So können Methoden Daten für die Klasse in Empfang nehmen, Feldinhalte (Daten) der Klasse ausgeben oder die Feldinhalte der Klasse be- bzw. verarbeiten.[30]

Wenn wir allgemein von Autos sprechen, so ist „Auto“ die Klasse. Autos sind charakterisiert durch Marke/Typ, Farbe, Bereifung, Leistung, etc. Betrachten wir ein bestimmtes Auto, eine „Instanz der Klasse Auto“, so können wir diesen Eigenschaften konkrete Werte zuweisen: Ein konkretes Auto ist beispielsweise ein blauer Opel Zafira mit 85 kW Leistung etc.

- Klassen werden mit dem Schlüsselwort `class` definiert.
- Bestandteile (*member*) werden innerhalb der Klasse mit `[modifiers] type identifier;` definiert.

[29]VC2012: S. 97.
[30]VC2012: S. 97.

- Modifier [VC:8.2]
- Instanzen werden mit dem Schlüsselwort `new` erzeugt. [VC:8.2a]

C# arbeitet (wie Java, im Gegensatz zu C++) ausschließlich mit Referenzen (=Pointer) auf Objekte. Der Rückgabewert einer Instanzerzeugung mit `new` ist also eine Referenz auf ein Stückchen Speicher im Freispeicher (Heap). Die Freigabe des Freispeichers erfolgt automatisch (garbage collection, GC). Bei der Zuweisuns von Objekten wird lediglich die Referenz kopiert.

Abb. 10 zeigt diesen Vorgang einer Zuweisung. Nach der Zuweisung zeigen beide Variablen auf dasselbe Objekt. Das Fahrzeug mit der Geschwindigkeit 86 könnte nun vom GC aus dem Freispeicher entfernt werden.

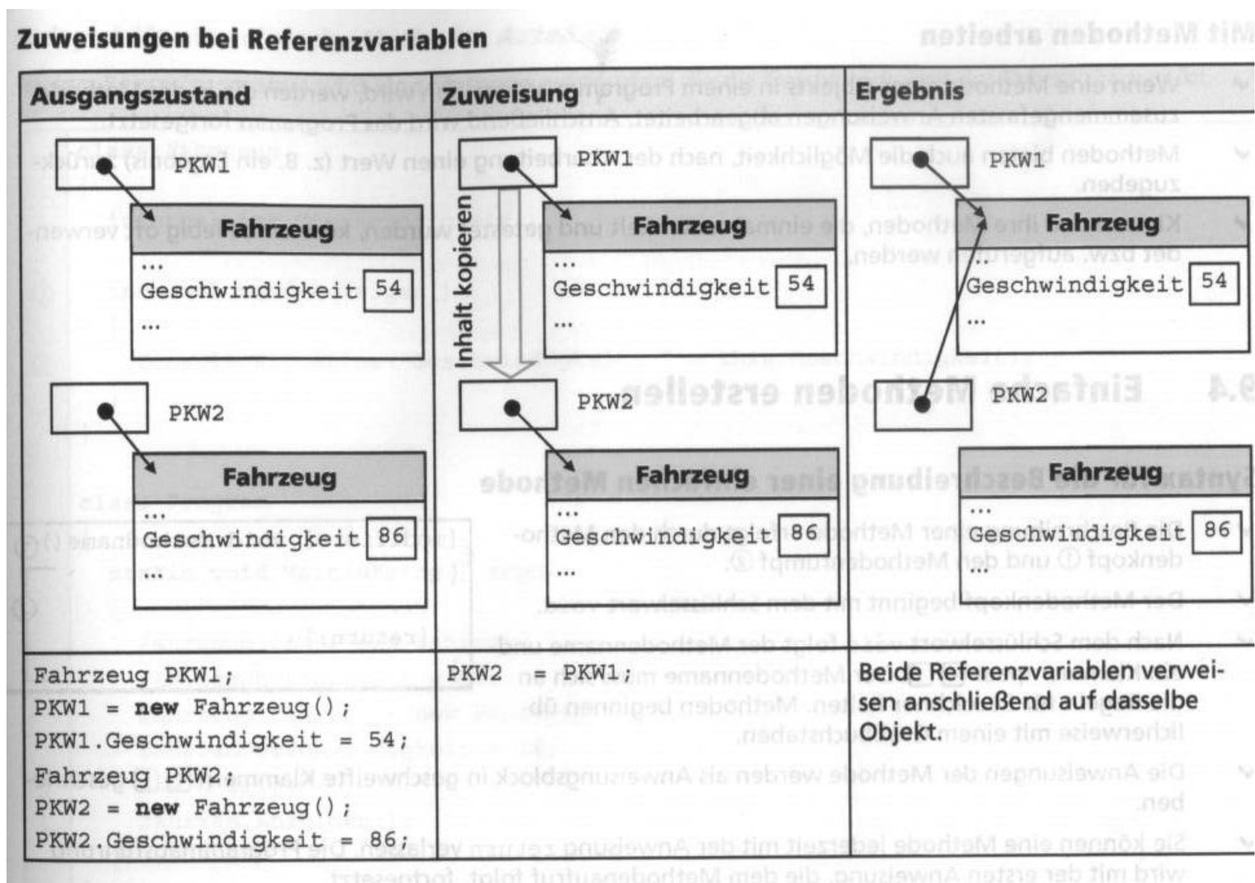


Abbildung 10: Kopieren von Instanzreferenzen[31]

Methoden mit Parametern erstellen

In der Klassendefinition werden die Methoden definiert:

```
[modifiers] void Methodenname([ref|out] type1 identifier1 [= default] [, ...]) {
    ...
}
```

- Bei nicht-elementaren Typen (Klassen) ist der Identifier immer eine Referenz.

[31]VC2012: S. 101.

- `ref/out` bezieht sich in diesem Fall auf die Referenz, nicht auf das Objekt!

Rückgabewerte werden wie bei statischen Methoden angegeben. Auch hier ist zu beachten, dass bei nicht-elementaren Typen eine Referenz auf eine Objektinstanz zurückgegeben wird.

Bei objektorientierten Sprachen können Methoden mit demselben Namen, aber unterschiedlichen Signaturen (= Übergabeparametern) definiert werden. Der Compiler wählt die richtige Implementierung [VC:8.9].

Aufgabe:

Übung 8.12.3: Überladen einer Methode

5.2 Kapselung und Konstruktoren[32]

Kapselung

Auf Felder einer Klasse soll grundsätzlich nur kontrolliert zugegriffen werden. In der Klasse muss die Möglichkeit bestehen, eine Wertptüfung o.ä. vorzunehmen. Dies führt in anderen oo Sprachen (Java, C++) zu folgenden stereotypen Konstrukten:

```
private int wert; // Feld ist von aussen nicht zugreifbar
public void setWert(int i) { this.wert = i; }
public int getWert() { return this.wert; }
```

Mit dieser Konstruktion hat der Entwickler die Möglichkeit, eine Wertprüfung zu ergänzen, ohne die Schnittstelle nach außen zu verändern.

C# hat hier eine elegantere Möglichkeit. Das Feld selbst wird zwar von außen zugreifbar gemacht. Prüfungen können, wie im Folgenden gezeigt, ergänzt werden.

Eigenschaften

Das Konzept der Eigenschaften sieht vor, dass diese in der Verwendung wie öffentliche Membervariablen aussehen:

```
// set
Instance.Member = value;
// get
var = Instance.Member;
```

Soll der Zugriff also ohne weitere Prüfung oder Einschränkung erfolgen, so kann die Membervariable so definiert werden:

```
class MeineKlasse {
  public Type Member;
  ...
}
```

[32] VC2015: Kap. 9

Soll das beispielsweise Schreiben der Variable nicht-öffentlich sein, so kann dies so formuliert werden:

```
class MeineKlasse {
  public Type Member { get; private set; }
  ...
}
```

Soll beim Schreiben zusätzlich eine Prüfung statt finden, muss zusätzlich eine interne Repräsentation der Membervariablen angelegt werden:

```
class MeineKlasse {
  private Type member; // interne Repr"asentation
  public  Type Member {
    get {
      return member;
    }
    private set {
      if ( value < 0 ) ... // Wertpr"ufung
      member = value;
    }
  }
  ...
}
```

Der Name `value` ist dabei durch die Sprache vorgegeben.
Ein komplettes Beispiel: [VC:9.2].

Konstruktoren

- Konstruktoren sind spezielle Methoden einer Klasse.
- Ein Konstruktor wird beim Erzeugen einer Instanz aufgerufen (`new ...`).
- Das Entfernen einer Instanz aus dem Speicher erfolgt nicht deterministisch (*garbage collector*).
- Grundsätzlich existiert der Standardkonstruktor (Konstruktor ohne weitere Parameter).
- Selbstdefinierte Konstruktoren unterscheiden sich in der Parameterleite (=Signatur) [VC:9.3]
- Ein Kontruktor kann einen anderen Konstruktor derselben Klasse verwenden (s. Abb. 11).

Objektinstanzen können auch über *Objektinitialisierer* mit Werten belegt werden [VC:9.3a, 9.3b].

Aufgabe (Bücherei):

- Erstellen Sie eine Klasse `Nutzer` mit den entsprechenden Bestandteilen, dazu einen Konstrutor mit den drei Parametern und eine `ToString`-Methode.
  `public override string ToString()`

[33]VC2012: S. 125.

```
...
public Fahrzeug(int wert)
{
  this.Geschwindigkeit = wert;
}

public Fahrzeug()    ②
  : this(50)         ①
{
}
...
```

Abbildung 11: Konstruktoren[33]

- Ergänzen Sie die Klasse `Nutzerverwaltung` um eine Methode `internal static Nutzer Lesen(int Nutzerid)`, die aus der Datei `nutzer.csv` die Zeile `Nutzerid` liest, daraus eine Instanz von `Nutzer` anlegt und zurückgibt. Falls `Nutzerid` größer als die Anzahl der Zeilen ist, wird `null` zurückgegeben.
- Rufen Sie `Lesen()` im Hauptprogramm auf. Prüfen Sie den Rückgabewert auf `null` und geben Sie, falls vorhanden, den Nutzer aus.

Statische Klassen[34]

Statische Eigenschaften sind Daten, sie nur einmal im System vorhanden sind und bereits mit dem Laden des Programms angelegt werden [VC:9.4].

Enthält eine Klasse nur statische Elemente, so kann die ganze Klasse als statisch deklariert werden. Dies hat zur Folge, dass von einer solchen Klasse keine Instanzen (`new` ...) gebildet werden können.

Partielle Klassen[35]

Die Bildung von Rahmenwerken oder die Entkopplung von Softwareteilen macht es oft nötig, dass

- Teile einer Klasse generiert, andere Teile von Hand ergänzt werden.
- Teile einer Klasse in anderen Softwareteilen sichtbar sein sollen, andere nicht.

[34]VC2015: Kap. 9.4
[35]VC2015: Kap. 9.6, 9.7

Bisher wurden solche Probleme durch Vererbung gelöst. Mit C# hat der Entwickler nun die Möglichkeit Klassen auf mehrere Dateien aufzuteilen. Um dies dem Compiler mitzuteilen, muss die Klasse als `partial class` definiert werden.

Darüber hinaus gibt es in partiellen Klassen die Möglichkeit, Methoden in einem Teil nur zu deklarieren, in einem anderen Teil hingegen zu definieren. Fehlt in einem Programmteil die Definition, wird die Ausführung der Methode weggelassen [VC:9.7].

> Aufgabe:
>
> Übung 9.9.2: Kapselung - Zugriff auf Felder über Eigenschaften

5.3 Vererbung[36]

> Die Vererbung ist ein wichtiges Merkmal objektorientierter Programmiersprachen. ...
>
> - Eine Klasse kann nur von *einer* Klasse abgeleitet werden (*Einfachvererbung*).[37]

Die Syntax für die Vererbung lautet:

```
[modifier] class Klasse : Basisklasse
{
   ...
}
```

Soll in einem Konstruktor der abgeleiteten Klasse der Konstruktor der Basisklasse aufgerufen werden, geschieht das mit `: base()`.

Alle Klassen erben direkt oder indirekt von `System.Object`. Von dieser Klasse erben alle Klassen die Methoden `GetType()` und `ToString()`.

> Aufgabe (Bücherei):
>
> - Leiten Sie von der Klasse `Medium` die Klassen `Buch` und `CD` ab (kann in derselben Datei erfolgen).
>   - Verteilen Sie die Attribute sinnvoll auf die Klassen.
>   - Ergänzen Sie jede Klasse um eine Methode `string Format()`, die den Datensatz als Zeichenkette in folfendem Format zurückgibt:
>   `<Signatur>,<Autor>,<Titel>,<Typ>,<Seitenzahl>,<Spieldauer>`
>   Für die jeweilige Klasse nicht zutreffende Werte werden mit Dummy-Werten belegt. In den abgeleiteten Klassen verlangt der Compiler ein zusätzliches `override`.
>   - Der Konstruktor von `Buch` wird mit den Parametern
>   `signatur, autor, titel, seitenzahl` aufgerufen.

[36] VC2015: Kap. 10
[37] VC2012: S. 138.

- Der Konstruktor von `CD` wird mit den Parametern `signatur, autor, titel, spieldauer` aufgerufen.
- Beide Konstruktoren verwenden den Konstruktor von `Medium`, der die gemeinsamen Attribute übernimmt.

- Legen Sie die Klasse `Medienverwaltung` mit folgenden Prozeduren an:
  - `Erzeugen` nimmt die sechs Paramter und erzeugt je nach Typ ein `Medium`, `Buch` oder `CD`.
  - `Anlegen(Medium)` öffnet die Datei `medien.csv` zum Anhängen und fügt über die `Format`-Methode einen Datensatz hinzu.
- Implementieren Sie im Hauptprogramm das Anlegen eines Mediums.

5.4 Polymorphismus

Wenn Sie nun über das Hauptprogramm Medien anlegen, werden Sie feststellen, dass in der Datei nur Datensätze vom Typ Medium erscheinen. An dieser Stelle wird nun ein Mechanismus benötigt, der zur Laufzeit die tatsächliche Klasse einer Instanz bestimmt und danach die passende Implentierung auswählt. Dieser Mechanismus nennt sich *Polymorphismus* (Vielgesichtigkeit).

Das polymorphe Überschreiben von Methoden erfolgt duch die *modifier* `virtual` in der Basisklasse und `override` in den abgeleiteten Klassen.

Aufgabe (Bücherei):

- Implementieren Sie die `Format()`-Methode polymorph.
- Überprüfen Sie erneut die Dateieinträge.

5.5 Abstrakte Klassen und Methoden[38]

Abstrakte Klassen implementieren ihre Methoden und Eigenschaften nur teilweise oder auch gar nicht. Die Implementiereung der abstrakten Methoden muss in den abgeleiteten Methoden erfolgen. Von abstrakten Klassen können keine Instanzen gebildet werden. Mithilfe abstrakter Klassen können Sie eine gewisse Grundfunktionalität für alle abgeleiteten Klassen zur Verfügung stellen. Außerdem können Sie erzwingen, dass bestimmte Elemente in den abgeleiteten Klassen implementiert und angepasst werden. [39]

[38] VC2015: Kap. 11.5, Wurm: 7.1.3
[39] VC2012: S. 155.

Abstrakte Klassen und Methoden erhalten das Schlüsselwort `abstract`. Eine Klasse muss als abstrakt gekennzeichnet werden, wenn mindestens eine Methode abstrakt ist ([VC:11.5], [Wurm: S. 212ff.]).

> Aufgabe:
>
> Ergänzen Sie die Bibliotheksanwendung:
>
> - Verwandeln Sie `MediumBase` in eine abstrakte Klasse mit `Format()` als abstrakte Methode.

5.6 Schnittstellen[40]

- Schnittstellen enthalten im Gegensatz zu abstrakten Klassen niemals Implementierungen der einzelnen Member, sondern nur deren Deklaration.
- Klassen ... können beliebig viele Schnittstellen implementieren.[41]

Syntax:

```
[public|internal] interface Iname [: I1 [, I2, ...]] {
    ...
}
```

Implementiert eine Klasse eine bestimmte Schnittstelle, so lautet die Syntax dafür:

```
class Klasse : [Basisklasse, ] I1 [, I2, ...]] {
    ...
}
```

Damit eine Klasse vollständig implementiert ist und damit instanziert werden kann, müssen alle übernommenen Schnittstellenmethoden implementiert werden ([VC:12], [Wurm: S. 218ff.]).

Anmerkung: Klassen, die die Schnittstelle `IDisposable` implementieren, können wie im Codebeispiel in Kap. 3 gezeigt, verwendet werden. Um das `IDisposable` zu implementieren, brauchen die Klassen eine `Dispose()`-Methode. Diese führt bei den Klassen zur Dateibearbeitung das Schließen der Datei aus.

> Aufgabe:
>
> [VC2015] Übung 12.6.2: Einfache Kontoführung

> Aufgabe (Bücherei):
>
> - Definieren Sie die Schnittstelle `IFormatable`, die eine `string Format()` vorgibt.
> - `Medium` soll diese Schnittstelle implementieren (erben).

[40] VC2015: Kap. 12, Wurm: Kap. 7.1.14
[41] VC2012: S. 164.

- Schreiben Sie in `IFormatable.cs` eine Funktion
  `static public void AddObjectToFile(IFormatable Object, string file)`,
  die die Datei zum schreiben öffnet und das Objekt über den Aufruf von `Format()` anhängt.
- Verwenden Sie in `Medienverwaltung.Anlegen` die neue Funktion.

6 Komplexe Datentypen[42]

6.1 Eindimensionale Arrays

Syntax für die Deklaration + Instanzierung + Initialisierung:

```
type[] var [ = new type[size] | = { wert1, wert2, ... } ] ;
```

- Der Elemetzugriff erfolt in der Form: `var[idx]`.
- Die Felder haben eine konstante Länge.
- Fehlerhafte Zugriffe führen zu einer `IndexOutOfRangeException`.

Wie bei Objekten ist in der Variable eine Referenz in den Freispeicher abgelegt (s. Abb. 12).

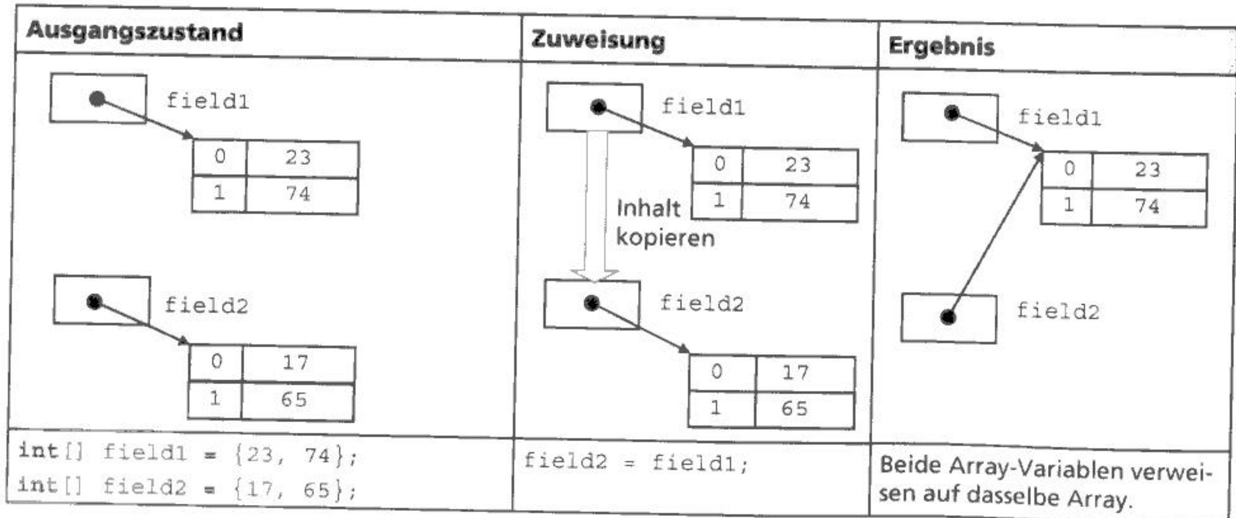


Abbildung 12: Zuweisung bei Array-Variablen[43]

Neben der Schleife über den Index (`for ( int i = 0, ...`) bietet C# die `foreach`-Schleife:

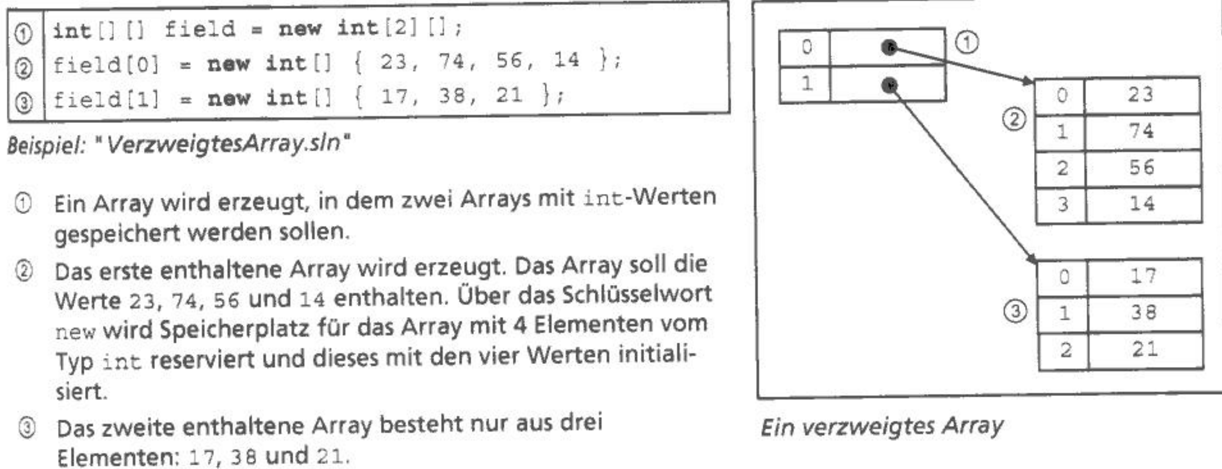
```
foreach ( type elem in ArrVar ) {
  ...
}
```

[42]VC2015: Kap. 13
[43]VC2012: S. 125.

6.2 Mehrdimensionale Arrays

Mehrdimensionale Arrays [VC:13.2]]

Verzweigte (unregelmäßige) Arrays: Werden in der 1. Dimension Referenzen auf weitere Arrays abgelegt, müssen diese nicht notwendig gleich lang sein (s. Abb. 13).



```
① int[][] field = new int[2][];
② field[0] = new int[] { 23, 74, 56, 14 };
③ field[1] = new int[] { 17, 38, 21 };
```

Beispiel: "VerzweigtesArray.sln"

① Ein Array wird erzeugt, in dem zwei Arrays mit int-Werten gespeichert werden sollen.

② Das erste enthaltene Array wird erzeugt. Das Array soll die Werte 23, 74, 56 und 14 enthalten. Über das Schlüsselwort new wird Speicherplatz für das Array mit 4 Elementen vom Typ int reserviert und dieses mit den vier Werten initialisiert.

③ Das zweite enthaltene Array besteht nur aus drei Elementen: 17, 38 und 21.

Ein verzweigtes Array

Abbildung 13: Verzweigte Arrays[44]

Da Arrays in C# besondere Klassen sind, gibt es auch Methoden dafür [VC:13.3].

Eine besondere Verwendung von Arrays ist das Parameter-Array. Damit ist es möglich Methoden zu definieren, die eine variable Anzahl von Paramtern (gleichen Typs) entgegen nimmt [VC:13.4].

6.3 Auflistungen

> Das .NET Framework stellt verschiedene Klassen für Auflistungen bereit, um beispielsweise Listen, Warteschlangen, Stapel, Wörterbücher oder sogenannte Hashtabellen zu erstellen. ... Sie müssen dazu den Namespace System.Collections zu Beginn einbinden[45].

Wichtig ist es zu wissen, welche Grundtypen an Auflistungen es gibt. Benötigt man eine davon, schaut man die Verwendung nach.

Liste → `ArrayList`, Stapel → `Stack`, Wörterbuch → `Dictionary`, Hashtabelle → `Hashtable`

> Alle Auflistungen und Arrays implementieren die Schnittstelle IEnumerable. Dadurch entsteht ein Positionszeiger, ein sogenannter Enumerator, zur Verfügung, mit dessen Hilfe Sie einen iterativen Lesezugriff auf die Auflistungs- oder Arrayelemente erhalten. Im Unterschied zur foreach-Schleife lässt sich der Enumerator mit der Methode

[44] VC2012: S. 125.

[45] VC2012: S. 183.

`GetEnumerator()` ermitteln, mit `Reset()` jederzeit auf den Anfang der Liste zurücksetzen und mit `MoveNext()` zum jeweils nächsten Element bewegen[46].

Beispiel: [VC:13.8]

`foreach` verwendet auch den Enumerator. Jede Klasse, die die `IENumerable`-Schnittstelle implementiert, lässt sich mit `foreach` durchlaufen.

Beispiel: [VC:13.8a]

Grundsätzlich lassen sich bei den Aufzählungstypen getypte von ungetypten unterscheiden. Bei ungetypten Aufzählungstypen werden einfach *Objekte* hintereinander abgelegt (z.B. `ArrayList`). Bei getypten Aufzählungstypen wird festgelegt, von welchem Typ die Objekte sein müssen, die in einer Aufzählung abgelegt werden (z.B. `Dictionary`).

Beispiel für `Dictionary`:

```
using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        // Create a new dictionary of strings, with string keys.
        //
        Dictionary<string, string> openWith =
            new Dictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // The indexer can be used to change the value associated
        // with a key.
        openWith["rtf"] = "winword.exe";

        // If a key does not exist, setting the indexer for that key
        // adds a new key/value pair.
        openWith["doc"] = "winword.exe";

        // When a program often has to try keys that turn out not to
        // be in the dictionary, TryGetValue can be a more efficient
        // way to retrieve values.
        string value = "";
        if (openWith.TryGetValue("tif", out value))
        {
            Console.WriteLine("For key = \"tif\", value = {0}.", value);
        }
        else
        {
            Console.WriteLine("Key = \"tif\" is not found.");
```

[46] VC2012: S. 186.

```
        }

        // ContainsKey can be used to test keys before inserting
        // them.
        if (!openWith.ContainsKey("ht"))
        {
            openWith.Add("ht", "hypertrm.exe");
            Console.WriteLine("Value added for key = \"ht\": {0}",
                openWith["ht"]);
        }

        // When you use foreach to enumerate dictionary elements,
        // the elements are retrieved as KeyValuePair objects.
        Console.WriteLine();
        foreach( KeyValuePair<string, string> kvp in openWith )
        {
            Console.WriteLine("Key = {0}, Value = {1}",
                kvp.Key, kvp.Value);
        }
    }
}
```

6.4 Aufzählungstypen

Im Gegensatz zu C/C++ lässt sich zu einem Element eines Aufzählungstypen auch dessen sprechender Name ermitteln (s. Abb. 14)

Aufgabe:

[VC2015] 13.17: Enumerationen

Aufgabe (Bücherei):

- Ändern Sie die Klasse `Medienverwaltung` in eine statische Klasse, die in ihrem Konstuktor `medium.csv` liest, für jede Zeile über `Erzeugen` ein Medium anlegt und dieses dem Verzeichnis `MediaDict` hinzufügt: Schlüssel ist die Signatur, Wert die Instanz.
- Implementieren Sie in `Medienverwaltung` die Methode `Lesen(String Signatur)`, die prüft, ob in `MediaDict` die Signatur vorhanden ist. Falls ja, wird die entsprechende Medieninstanz, falls nein `null` zurückgegeben.
- `AlleLesen` bildet aus dem Verzeichnis eine Liste aller Medien und gibt diese zurück.
- `Anlegen` muss um das Neuladen von `MediaDict` ergänzt werden.
- Implementieren Sie im Hauptprogramm die Funktionen *Lesen* und *Alle lesen* für die Entität *Medium*. Die Ausgabe soll jeweils unter Verwendung von `Format()` erfolgen.

[47] VC2012: S. 195.

```
① enum Groesse
  {
②   Klein = 5,
    Mittelgross = 10,
③   Gross = Klein + Mittelgross
  }

  class Program
  {
    static void Main(string[] args)
    {
④     Console.WriteLine("Wert von {0}: {1}", Groesse.Gross, (int)Groesse.Gross);
⑤     Console.WriteLine("Wert von {0} + 1: {1}", Groesse.Gross, 1 + Groesse.Gross);
⑥     foreach (string s in Enum.GetNames(typeof(Groesse)))
      {
        Console.WriteLine(s);
      }
⑦     foreach (int i in Enum.GetValues(typeof(Groesse)))
      {
        Console.WriteLine(i);
      }
    }
  }
```

Abbildung 14: Aufzählungstypen[47]

6.5 Generische Typen

Wie in Abschnitt 6.3 gezeigt, können Auflistungen getypt und ungetypt implementiert sein. Bei getypten Auflistungen muss der aktuelle Typ bei der Instanzierung in spitzen Klammern mit angegeben werden (z.B. `Dictionary <string ,string>`). Solche generischen Typen können auch selbst implementiert werden. Als Beispiel sei eine generische `Stack`-Implementierung gezeigt[48]:

```
public class Stack<T>
{
    ...
    T[] m_Items;
    public Stack():this(100) {}
    public Stack(int size)
    {...}
    public void Push(T item)
    {...}
    public T Pop()
    {...}
}
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
int number = stack.Pop();
```

[48] https://msdn.microsoft.com/en-us/library/ms379564(v=vs.80).aspx (11.7.2017)

Aufgabe:

Implementieren und testen Sie dieses Beispiel.

7 Fehlerbehandlung[49]

Zur Fehlerbehandlung gibt es verschiedene Ansätze:

- Rückgabewerte vom Typ `int`
- Rückgabewerte eines speziellen Fehlertyps
- Exceptions
- Eigene Routine zur Ermittlung des Fehlerstatus

Vorteil der Rückgabewerte:

- Einfachere lokale Fehlerbehandlung

Vorteil der Exceptions:

- Fehlerbehandlung stört nicht den logischen Programmfluss.
- Fehlerbehandlung wird nicht vergessen.

Empfehlung:

- Zu erwartende (oft fachliche) Fehler werden auf Rückgabewerte abgebildet.
- Unerwartete Fehler (oft technische Fehler, Logikfehler) werden über Exceptions behandelt.

```
throw new ExceptionName("Ursache");
```

Exception dürfen nicht zum Anwender vordringen. Sie müssen vorher abgefangen werden:

```
try {
   ...
}
catch (ExceptionName e) { // fangt spezielle Exception
   ...
}
catch (Exception e) { // fangt den Rest
   ...
}
```

Exception-Klassen können auch selbst definiert werden [VC:14.3].

Aufgabe (Bücherei):

Folgende Fehlersituationen sollen abgefangen werden:

[49] VC2015: Kap. 14

- Eingabe einer bereits vorhandenen Signatur:

  `Medienverwaltung.Anlegen` bekommt einen `int`-Returncode:

  - Definieren Sie passende `int`-Konstanten.
  - Geben Sie bei einem Duplikat den entsprechenden `int`-Wert an das Hauptprogramm zurück.
  - Melden Sie im Hauptprogramm an den Anwender zurück, ob das Medium erfolgreich angelegt werden konnte.

- Fehlerhaftes Dateiformat:

  - Prüfen Sie nach den `split`-Operationen in `Medienverwaltung` und `Nutzerverwaltung`, ob sechs bzw. drei Elemente aus der Zeile extrahiert werden konnten (`array.Length`). Falls nein, werfen Sie eine Ausnahme:
    ```
    throw new ApplicationException("Unerwartete Datensatzlange");
    ```
  - Fangen Sie im Hauptprogramm die Ausnahme.

8 Spezielle Themen

8.1 XML-Serialisierung

Objektdaten müssen gelegentlich zwischengespeichert werden. Dazu müssen sie ein eine Form umgewandelt werden, die keine Zeiger enthält. Dies leistet eine *Serialisierung*. Ist das Speicherformat eine XML-Darstellung, spricht man von XML-Serialisierung. Ein anderes dafür gerne verwendetes Format ist JSON[50].

XML-Serislisierung[51]

Aufgabe:

Bringen Sie das gezeigte Codebeispiel zum Laufen

Aufgabe (Bücherei):

Die Verwaltung von Ausleihen soll über eine XML-Serialisierung erfolgen. Legen Sie dazu die Klasse `Ausleiheverwaltung` an:

- Im Konstruktor wird geprüft, ob die Datei `ausleihen.xml` vorhanden ist. Falls ja, wird sie gelesen, deserialisiert und der Membervariablen `Dictionary<String,Nutzer> Ausleihen` gespeichert.

[50] Java Script Object Notation

[51] https://www.jonasjohn.de/snippets/csharp/xmlserializer-example.htm (7.12.2022)

- Im `Dispose` wird `Ausleihen` serialisiert und in der Datei gespeichert.
- Die Funktion `int Anlegen(String Signatur, int Nutzernummer)`
  - prüft über die entsprechenden `Lesen`-Funktionen, ob die Parameter gültig sind,
  - prüft, ob das genannte Medium nicht schon ausgeliehen ist (die Signatur in `Ausleihen` als Schlüssel vorhanden ist),
  - fügt den `Nutzer` `Ausleihen` hinzu.
  - Im Fehlerfall wird ein entsprechender Fehlercode zurückgegeben.
- Die Funktion `int Enfernen(String Signatur, int Nutzernummer)` prüft, ob das Medium ausgeliehen ist und entfert es ggf. aus `Ausleihen`.
- Legen Sie über `using` ein Objekt der `Ausleiheverwaltung` im Hauptprogramm an und implementieren Sie die Funktionen *Anlegen* und *Löschen*.

8.2 Delegaten und Ereignisse

Delegaten

Delegaten sind die C#-Variante von Funktionspointern. Dieses aus C stammende Konstrukt dient dazu, eine Implementierung zur Laufzeit festzulegen. Im Abschnitt 5.4 wurde bereits gezeigt, dass dieses Ziel auch mit Mitteln der Objektorientierung erreicht werden kann. Delegaten sind die Standardmethode ausführbaren Code an Ereignisse zu hängen.

Die Verwendung geschieht in drei Schritten:

Als ersten muss ein Typ für die gewünschte Signatur erstellt werden:

```
public delegate <type> <name> (<params>);
```

Beispiel: `public delegate void Del(string message);`

Als zweites muss eine statische Funktion mit dieser Signatur existieren.

Beispiel:

```
public static void DelegateMethod(string message)
{
    System.Console.WriteLine(message);
}
```

Als drittes wird der Delegat instanziert und verwendet.

Beispiel:

```
// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Schritt zwei und drei können durch die Verwendung von Lambda-Funktionen (anonyme Funktionen) zusammengefasst werden. Für die Definition von Labmbda-Funktionen gibt es zwei Varianten:

Lambda-Ausdruck: `(input parameters) => expression`

Beispiel[52]:

```
delegate int del(int i);
static void Main(string[] args)
{
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

Lambda-Statement: `(input parameters) => {statement;}`

Beispiel:

```
delegate void TestDelegate(string s);
...
TestDelegate myDel =
  n => { string s = n + " " + "World"; Console.WriteLine(s); };
myDel("Hello");
```

Ereignisse[53]

Gerade in der GUI-Programmierung wird viel mit Ereignissen umgegangen. Jede Aktion auf dem Bildschirm führt zu einem Ereignis, das entsprechend behandelt werden muss. C# stellt zur Ereignisbehandlung einen Standarddelgaten zur Verfügung:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Ein Ereignis wird nun so definiert:

```
public event EventHandler MyEvent;
```

Anmelden eines Eventhandlers:

```
public void MyHandler(object sender, EventArgs e) {
  // do anything
}

MyEvent += new EventHandler(MyHandler);
```

Auslösen der Handler:

```
// Check, if there is any Handler
if (MyEvent != null)
  MyEvent(this, null);
```

[52]MSDN: Lambda Expressions (C# Programming Guide)
[53]Wurm: Kap.7.2

Aufgabe (Bücherei):

Rückgabewerte vom Typ `int` haben den Nachteil, dass kein fehlerspezisfischer Text transportiert werden kann. Eine Möglichkeit dies zu umgehen, ist es, einen Informationskanal einzurichten, über den entsprechende Probleme reportiert werden können. Im Beispiel *Ausleihe anlegen* kann sowohl die Nutzernummer als auch die Signatur fehlerhaft sein, als auch das Medium bereits ausgeliehen sein. Es ist nicht von Vorteil für alle erdenklichen Fälle Fehlercodes einzurichten. Fehlercodes sollten die grundlegenden Probleme wie `RC_NOT_FOUND` oder `RC_EXISTS` abbilden. Was aber nicht da ist oder schon da ist, kann dann über den Informationskanal weitergegeben werden.

Ein Informationskanal gehört zur technischen Infrastruktur und nicht zur Fachlichkeit, sollte daher nicht mit in die Schnittstelle von `Anlegen` mit aufgenommen werden. Eine Möglichkeit, dies zu umgehen, ist folgende:

- Schreiben Sie eine statische Klasse `Infokanal` mit folgenden Eigenschaften:
  - Ein `event EventHandler Events` als *member*,
  - einer Methode `Anmelden(EventHandler Handler)`, die `Handler Events` hinzufügt,
  - einer Methode `Melden(EventArgs e)`, die dann die Handler aufruft.
- Rufen Sie in `Ausleiheverwaltung.Anlegen` bei den entsprechenden Fehlerfällen `Melden` auf.
- Schreiben Sie je einen Handler, der die Meldungen über die Konsole bzw. über ein Popup ausgibt.
- Melden Sie im Konsolenhauptprogramm den „Konsolenmelder" an, bevor Sie `Ausleiheverwaltung.Anlegen` aufrufen.
- Melden Sie bei der Initialisierung der GUI den „GUI-Melder" an.

9 Quellen

Codeguru	.NET / C#: http://www.codeguru.com/csharp/
CsharpEx	C# Examples: http://www.csharp-examples.net/
MSDN	MSDN-Internetseiten: https://msdn.microsoft.com
VC2012	Visual C# 2012. Grundlagen der Programmierung. Herdt 2013.
VC2015	Visual C# 2015. Grundlagen der Programmierung. Herdt 2015.
Yoda	Jon Skeet on the web: http://www.yoda.arachsys.com/