# Allgemeine Konzepte der OOP

https://xainey.github.io/2016/powershell-classes-and-concepts/

Die objektorientierte Programmierung (OOP) ist eine seit vielen Jahren eingeführte Art und Weise des Programmierens, die sich wesentlich von der sog. strukturierten Programmierung unterscheidet.

C:\Users\C\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\4DDEELXO\MC900351929[1].wmfDie strukt. Programmierung (auch prozedurale Programmierung genannt) trennt zwischen den Daten und den Methoden, die diese Daten manipulieren. Die objektorientierte Programmierung führt diese beiden Aspekte wieder in Form von Klassen zusammen. Die Klasse ist der Container für Daten, die logisch gesehen zusammengehören und einen realen/irrealen Gegenstand unseres Systems abbilden. Sie beinhaltet auch sämtliche Methoden, die diese Daten manipulieren sollen.

**Wichtige Konzepte**

|  |  |
| --- | --- |
| * Klasse / Objekt / Instanziierung * Attribute / Methoden * Konstruktor/Destruktor * Statische Attribute und Methoden * Geheimnisprinzip | * Einfach-, Mehrfachvererbung * Abstrakte Klasse / Interface * Überschreiben von Methoden/Polymorphie * Assoziation/Aggregation/Komposition |

### Klasse / Objekt /Instanziierung

Die Klasse ist der grundlegende Begriff in der OOP.

Ausgehend von der jeweiligen Problemstellung versucht man, die zusammengehörenden Informationen in einem gemeinsamen Container, der Klasse eben, zu speichern.

Die Klasse erhält einen Namen, die zu speichernden Informationen bezeichnet man als Attribute der Klasse.

Die Manipulation, d.h. das Schreiben, Lesen und Ändern der Daten, obliegt ebenfalls dem Verantwortungsbereich der Klasse. Neben diesen Aufgaben kann eine Klasse noch weitere Fähigkeiten besitzen. Diese Fähigkeiten werden in der Klasse durch Methoden definiert.

Eine Klasse besteht deshalb zumindest aus 3 Bereichen.

1. Dem Klassennamen
2. Der Liste der Attribute
3. Der Liste der Methoden

Ein typisches Beispiel für eine Klasse könnte wie folgt aussehen:

|  |  |
| --- | --- |
| Quellcode | Klassendiagramm |
| class Person{  $Alter  $Name  $Vorname  [string] getName()  {  return $this.Name  }  [void]setName([string] $VorName)  {  $this.Name = $VorName  }  [void]sprechen([string] $Text)  {  *<# HAs to be done#>*  }  } |  |

Neben den wirklichen Fähigkeiten, im obigen Beispiel sprechen, muss eine Klasse häufig über Verwaltungsmethoden verfügen, die eine Manipulation der internen Attribute ermöglichen. Man nennt diese Methode häufig getter/setter-Methoden. Da ein Methodenname lediglich einmal genutzt werden darf, muss man deshalb häufig 2 Methodennamen benutzen.

Innerhalb der Klasse gibt es häufig eine spezielle Methode, den sog. Konstruktor. Sie hat den gleichen Namen wie die Klasse selbst und sie kann in verschiedenen Variationen vorliegen. Beim Erzeugen eines Objektes (s.u.) wird diese Methode als Erstes aufgerufen. Wird diese Methode nicht innerhalb der Klasse definiert, so wird ein sog. Standardkonstruktor benutzt. Er besteht lediglich aus dem Methodennamen ohne irgendwelche Parameter. Werden eigene Konstrukoren geschrieben, so muss der Standardkonstruktor explizit definiert werden, sonst ist er nicht mehr vorhanden.

|  |  |
| --- | --- |
| Quellcode | Klassendiagramm |
| class Person{  $Alter  $Name  $Vorname  Person(){  }  Person([string]$FamName, [string]$Vorname, $Alter)  {  $this.Alter = $Alter  $this.Vorname = $Vorname  $this.Name = $FamName  }  [string] getName(){  return $this.Name  }  [void]setName([string] $VorName){  $this.Name = $VorName  }  [void]sprechen([string] $Text){  *<# HAs to be done#>*  }  } |  |

### Benutzung von Objekten / Konstruktor

Die Klasse definiert als Bauplan lediglich die Art der gehaltenen Informationen bzw. die Methoden, die zur Manipulation der Daten bzw. zur Funktionalität der Klasse notwendig sind. Doch wie kann man nun diese Klasse nutzen? Da wir ja Informationen zu einem realen Gegenstand des Systems erheben wollen, müssen wir den Bauplan der Klasse einem realen Objekt zuordnen. Nur ein reales Objekt kann Daten speichern. Dieser Vorgang wird Instanziierung genannt und läuft in folgenden Schritten ab:

* Deklaration einer Variablen vom Typ der Klasse
* Instanziierung des Objektes mit Hilfe des new-Operators.

. .\person.ps1

$karl = [Person]::new()

$gerd = [Person]::new("Gerd", "Sych", 76)

$gerd.Alter

$karl.Vorname

Der new-Operator erzeugt ein sog. Objekt der Klasse. Dieses Objekt ist einzigartig und ist nun in der Lage, die Informationen aufzunehmen und zu verarbeiten. Bei allen Instanziierungen wird immer ein Konstruktor aufgerufen. Er lautet wie der Name der Klasse und kann überladen sein, d.h. in verschiedenen Versionen existieren. Durch ihn ist das Objekt in der Lage, Zustände seiner Variablen bei der Erzeugung zu kontrollieren. Definiert man eigene Konstruktoren, so muss der parameterlose Standardkonstruktor ebenfalls angegeben werden, wenn man ihn zur Verfügung stellen will.

Erst nach dem Erzeugen kann man nun die Fähigkeiten des Objektes benutzen, d.h. man kann die Methoden der Klasse benutzen. Merke: Methoden werden auf Klassenebene definiert, aber auf Objektebene genutzt ! (Es gibt aber eine Ausnahme ! Welche ?)

Parallel zum Konstruktor gibt es den sog. Destruktor. Er wird durch den folgenden Aufruf beschrieben:

~Klassenname()

Das Zerstören eines Objektes wird durch das NULL-Setzen der Objektreferenz bewirkt. Der tatsächliche Zeitpunkt des Zerstören eines Objektes hängt von mehreren Faktoren ab

* Anzahl der noch gültigen Referenzen
* Tatsächliches Freigeben des Speichers durch die Garbage Collection

Es kann deshalb nicht genau vordefiniert werden, ob und wann durch das Löschen einer Objektreferenz die Destruktor-Methode aufgerufen wird.

Hinweis: In Powershell gibt es keinen Destruktor

Beispiel:

class Person{

$Alter

$Name

$Vorname

Person(){

}

Person([string]$FamName, [string]$Vorname, [int]$Alter)

{

$this.Alter = $Alter

$this.Vorname = $Vorname

$this.Name = $FamName

}

[string] getName()

{

return $this.Name

}

[void] setName([string] $VorName)

{

$this.Name = $VorName

}

[void] sprechen([string] $Texjght){

Write-Host "bla"

}

<#

~Person()

{

Write-Host "Good bye baby"

}

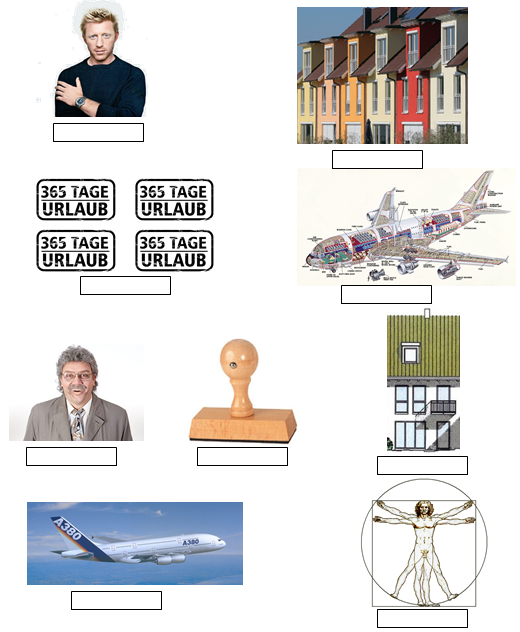
}

#>

### 

### Aufgabe:

Weisen Sie den untenstehenden Bildern jeweils die Begriffe Klasse bzw. Objekt zu



**C:\Users\C\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\4DDEELXO\MC900351929[1].wmfAufgabe:** Überlegen Sie sich, welche Klassen es in Ihrer beliebten Berufsschule gibt.

Finden Sie auch einige Attribute und Methoden

### Statische Attribute/Methoden

Um in OOP arbeiten zu können, sind offenbar immer zunächst Objektinstanziierungen notwendig. Dies ist aber manchmal lästig, weil wir eigentlichnur eine Funktionalität brauchen bzw. redundante Informationen speichern wollen, die für alle gleich sind.

So könnte es z.B. sein, dass Schüler einer Klasse die Anzahl der Mitschüler in ihrer Klasse kennen sollen. Bei 32 Schülern würde dies bedeuten, dass 32 Objekte die gleiche Information in einer lokalen Instanzvariable halten müssten! Noch schlimmer, bei Änderungen der Schülerzahl durch Hinzukommen / Weggehen neuer Schüler müsste in allen Objekten dieser Wert geändert werden !

Für solche Fälle kennt die OOP die Möglichkeit sog. statischer Attribute bzw. Methoden. Diese werden bei der Klasse gehalten und durch das Schlüsselwort static deklariert.

Der Zugriff auf diese Werte ist sowohl über jedes Objekt als auch über die Klasse an sich möglich.

Die Aktualisierung solcher Informationen beim Zerstören solcher Objekte kann aber problematisch werden.

Folgendes Beispiel verdeutlicht die Situation:

class Person{

[int] hidden $Alter

$Name

$Vorname

[int] static $AnzahlMitSchueler

Person(){

}

Person([string]$FamName, [string]$Vorname, [int]$Alter, [int]$MitSchueler)

{

$this.Alter = $Alter

$this.Vorname = $Vorname

$this.Name = $FamName

[Person]::AnzahlMitSchueler = $MitSchueler

}

[string] getName()

{

return $this.Name

}

[void] setName([string] $VorName)

{

$this.Name = $VorName

}

[void] sprechen([string] $Text)

{

Write-Host $Text

}

}

$Karl = [Person]::new("Steinam", "Karl", 20, 30)

$Karl.sprechen("blabala")

[Person]::AnzahlMitSchueler

$Gerd = [Person]::new()

$Gerd.sprechen("blabala")

$Gerd::AnzahlMitSchueler

$Gerd::AnzahlMitSchueler = 32

$Gerd::AnzahlMitSchueler

[Person]::AnzahlMitSchueler

$Karl::AnzahlMitSchueler

### Geheimnisprinzip

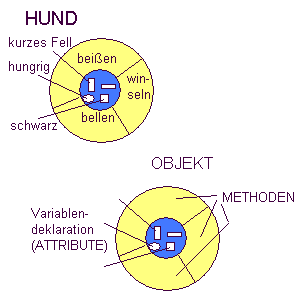
So wie Objekte im realen Leben auch nicht jedes Geheimnis nach draußen preisgeben, so gilt dies auch in der OOP. Das Objekt sollte prinzipiell den Zustand seiner Attribute versteckt halten, d.h. keinen direkten Zugriff auf seine Attribute erlauben.

Durch das Bereitstellen von entsprechenden getter/setter-Methoden bzw. Properties kann das Objekt den Zugriff auf seine Attribute kontrollieren.

Der generelle Zugriffsmöglichkeit auf Attribute und Methoden wird in der OOP über die Sichtbar-keiten definiert. Diese stehen vor dem Attribut bzw. der Methode und definieren den möglichen Zugriff von außerhalb auf die Attribute und Methoden. Falls keine Aussage im Quellcode getroffen wird, sol gelten die default-Sichtbarkeiten der jeweiligen Programmiersprachen, die sich durchaus unterscheiden können. Gängige Definitionen von Sichtbarkeiten sind:

* **public**: Öffentlich sichtbar, von überall aufruf- und damit manipulierbar
* **private**: Nur innerhalb des Objektes selbst benutzbar
* **protected**/internal: Nur innerhalb des gleichen Namespaces bzw. über Vererbungsmechanismen sichtbar.

…..

In den Bildern sind die Variablen des Objektes im Zentrum, umschlossen von den Methoden. Dies deutet grafisch an, daß Variablen gegen Zugriffe von außen geschützt sind, d.h. Programme können nicht direkt auf die Variablen des Objektes zugreifen, son-dern müssen die entsprechenden Methoden auf-rufen. Dieser Schutz wird Kapselung genannt. Mit Kapselung werden oft Implementationsdetails versteckt.

Ein Objekt hat eine nach außen definierte Schnitt-stelle, mit der andere Objekte kommunizieren können. Das Objekt kann die privaten Informa-tionen und Methoden ändern, ohne daß andere Objekte betroffen sind. Zudem kann ein Objekt problemlos herumgereicht werden.  
  
  
**Powershell kennt dieses Konzept nicht, das darunterliegende .NET-Framework allerdings.**

**Mit Hilfe des Schlüsselwortes hidden kann man zumindest den privaten Scope prinzipiell abbilden.**

**Arbeitsauftrag**

Erstellen Sie zwei Klassen (Auto, Mitarbeiter) zur Verwaltung eines Fuhrparks in einem Unternehmen  
Speichenr Sie sich für das Auto die Attribute (Hersteller, Modell, Leistung, Verbrauch, Baujahr, KilometerStand)  
Die Klasse Auto verfügt über eine Methode fahren(). Diese prüft, ob die Türen geschlossen sind. Falls ja, wird der Motar gestartet, falls nein erhält der Nutzer einen Piepston. Simulieren Sie den Piepston mit Hilfe von [console]::beep(Frequenz(190 - 8500, Dauer (ms))

Mitarbeiter können sich Fahrzeuge für eine Dienstreise ausleihen. Bei der Rückgabe müssen Sie die gefahrenen Kilometer sowie den Kilometerstand des Autos an den Chef melden.

Erstellen Sie zunächst ein Klassendiagramm

Implementieren Sie anschließend den Sachverhalt mit Hilfe von Powershell  
  
Der Mitarbeiter Brenner leiht sich für eine Dienstreise nach Hamburg (700 km einfach) den BMW mit dem Kennzeichen WÜ-MA-777 aus. Der Kilometerstand ist am Ende der Fahrt im Auto zu setzen.  
.

## Assoziationen zwischen Klassen

Assoziation mit Namensangabe, Leserichtung und Multiplizitätsangabe:

Assoziationen mit bidirektionaler Navigierbarkeit:

Verbot der Navigierbarkeit:

Unspezifizierte Navigierbarkeit:

Unidirektionale Navigierbarkeit:

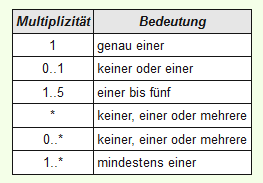
Teilweise spezifizierte Navigierbarkeit:

**Verwendung**

Ohne Assoziationen ist ein Klassendiagramm nur eine zusammenhanglose Ansammlung von Klassen. Erst durch die Modellierung der Assoziationen werden die Klassen zu einer funktionierenden Einheit.

Die folgenden Hinweise helfen Ihnen bei der Modellierung von Assoziationen:

* Klassen, die miteinander interagieren (sich kennen) müssen, um in der Gesamtheit die gewünschten Dienstleistungen zu erbringen, müssen durch Assoziationen verbunden werden.
* Die Interaktionen zwischen Klassen sollten durch Assoziationsnamen benannt werden.
* Das Verhältnis der Klassen während dieser Interaktionen wird durch die Rollenbezeichnungen definiert.
* Die Multiplizitäten legen fest, wie viele der aus den Klassen erzeugten Objekte miteinander interagieren müssen (dürfen).



**Arbeitsauftrag 1**

Entwerfen Sie ein Klassendiagramm mit vollständiger Assoziation zwischen den beiden Klassen Konto und Kunde. Eigenschaften und Methoden sind nicht erforderlich.

**Arbeitsauftrag 2**

In einem Unternehmen arbeiten Mitarbeiter. Die Mitarbeiter sind Abteilungen zugeordnet.

**Fragen**

In wie vielen Abteilungen kann ein Mitarbeiter arbeiten? Wie viele Mitarbeiter können in einer Abteilung arbeiten? Wie viele Abteilungen kann ein Mitarbeiter leiten?

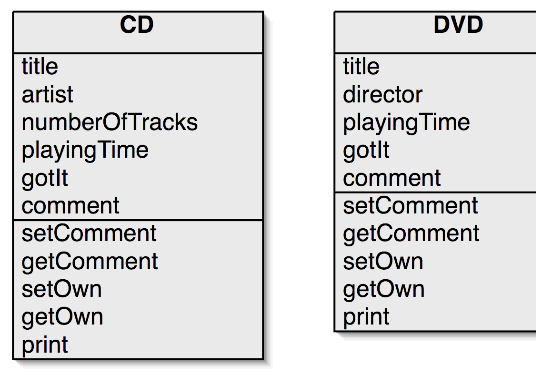
Von wie vielen Mitarbeitern kann eine Abteilung geleitet werden?

## Teil 2

## Der 2. Teil setzt sich mit folgenden Konzepten auseinander:

* Vererbung
  + Basisklasse / Abgeleitete KLasse
  + Einfachvererbung
  + Überschreiben von Methoden

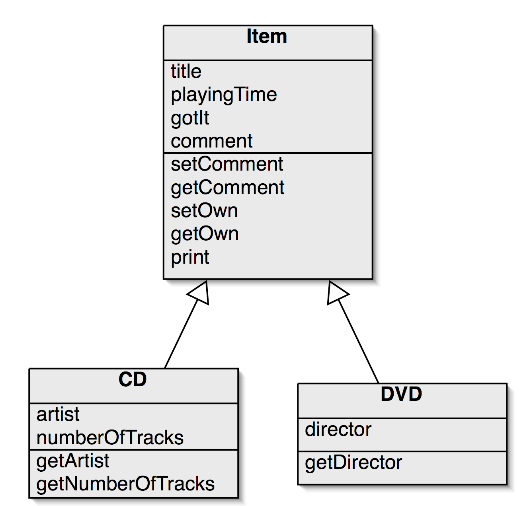
# Vererbung

Beim Erstellen von Klassen wird man häufig Abhängigkeiten/Gemeinsamkeiten zwischen Klassen erkennen. Irgendwie gehören sie zusammen aber doch auch wieder nicht. Die OOA löst diese Problematik durch das Konzept der Vererbung.

Es bedeutet, dass Klassen Attribute und Fähigkeiten anderer Klassen über-nehmen und gleichzeitig erweitern können.

Häufig wird man auch erst im Laufe der Analyse eine bestehende Klasse in mehrere Klassen unter-teilen wollen.

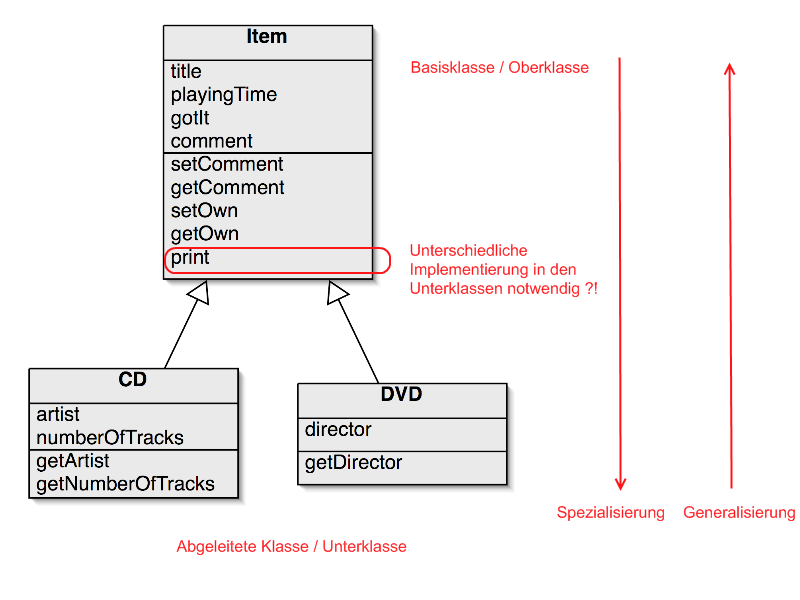
Aufgabe: Wie könnte man die dargestellten Klassen besser organisieren ?

Die Klasse CD und DVD verfügen über gemeinsame Attribute und Metho-den. Sie werden in einer Oberklasse Item ausgelagert. Zusätzliche Informa-tionen und Fähigkeiten bleiben in den spezialisierten Unterklassen.

Im Zusammenhang mit Vererbung existieren einige Begriffe:

* **Basisklasse**: Die Klasse, die alle Gemeinsamkeiten aufnimmt (in unserem Fall Item)
* **Abgeleitete Klasse / Unterklasse**: Die Klassen, die spezielle Attribute und Methoden aufnehmen (in unserem Fall CD und DVD)
* **Generalisierung**: Untersuchung des Vererbungsaspektes von den speziellen Klassen zu den allgemeinen Klassen

**Spezialisierung**: Untersuchung des Vererbungsaspektes von den allgemeinen Klassen zu den speziellen Klassen

Es sind folgende Dinge zu beachten:

* Informationen werden in Ober- und Unterklassen gehalten. Damit wird Redundanz vermieden.
* Die Unterklassen brauchen aber Zugriffsmöglichkeiten auf Attribute und Methoden der Oberklasse
* Die Unterklassen halten über den Vererbungsmechanismus den Vertrag der Oberklasse. Konkrete Instanzen der Unterklassen können überall dort benutzt werden, wo Instanzen der Oberklasse erwartet werden. In einem solchen Falle ist dann aber lediglich der Zugriff auf die Schnittstelle der Oberklasse möglich. Die Unterklassen sind somit vom gleichen Typ wie die Oberklasse. Eine DVD ist damit auch ein Item.
* Die Oberklasse hat im Allgemeinen keine Kenntnis darüber, dass es abgeleitete Unterklassen gibt.

## Implementierung des Beispiels in Powershell

Bei der Implementierung des obigen Beispiels ist zu beachten, dass

* insgesamt 3 Klassen erzeugt werden müssen
* die Unterklassen Zugriff auf Attribute und Methoden der Oberklasse haben müssen
* gewisse Methoden in den Unterklassen überschrieben werden müssen.

Implementierung der Oberklasse

|  |  |
| --- | --- |
| Quellcode | Kommentar |
| class Item  {  [string]$title  [int]$playingTime  [bool]$gotIt  [string]$comment  Item([string]$t, [int]$p){  $this.title = $t;  $this.playingTime = $p  }  [string]print(){  return $this.title + ", " + $this.playingTime + " Minuten"  }  } | Die Klasse Item hält 4 Attribute und 2 getter/setter-Methoden. Der Konstruktor wird verwendet, um Titel und Spielzeit zu setzen. Die print()-Methode gibt den Titel und die Spielzeit aus. |

### Implementierung der Unterklasse CD

Eine erste, naive, Implementierung sieht wie folgt aus:

class CD : Item

{

[string]$artist

[int]$numberOfTracks

CD([string]$a, [int]$n ){

$this.artist = $a

$this.numberOfTracks = $n

}

[string]print(){

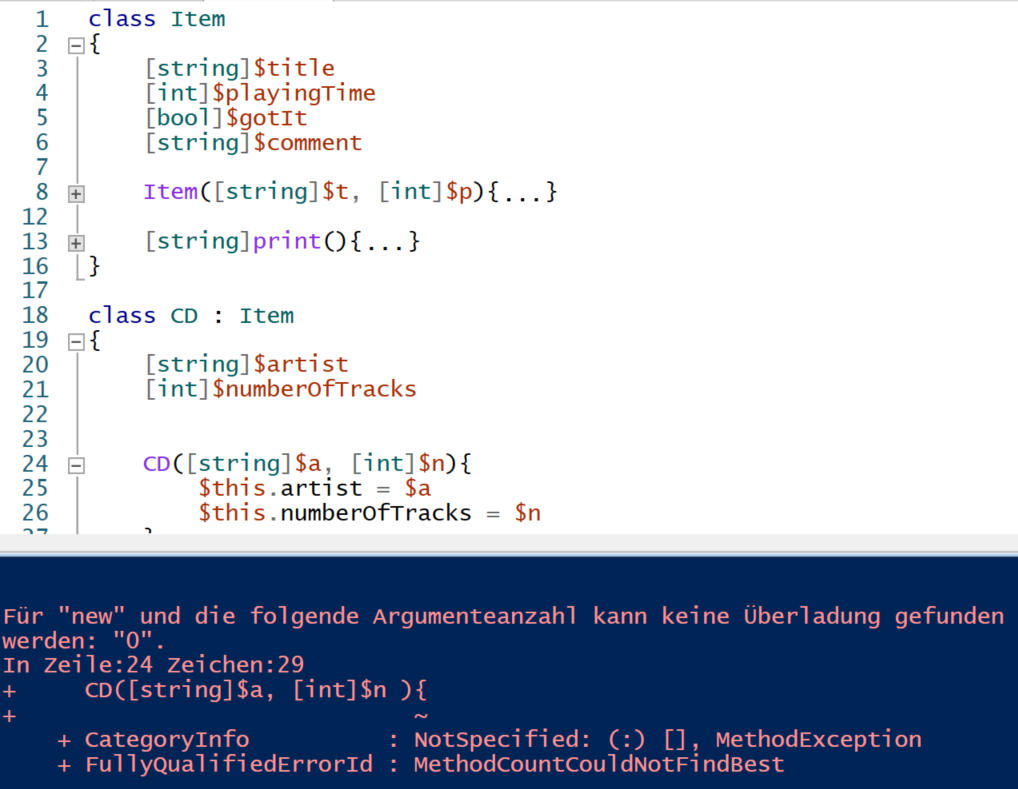
return $this.title + ", " + $this.artist + ", " `

+ $this.numberOfTracks + ", " + $this.playingTime + " Minuten"

}

}

Dies führt aber zu folgenden Problemen:



#### Konstruktorproblematik

Beim Erzeugen einer Unterklasse wird es aufgrund der Vererbung auch notwendig, eine Instanz der Oberklasse zu erzeugen. Da diese jedoch in unserer Implementierung keinen parameterlosen Konstruktor zulässt, erzeugt Powershell einen Fehler. Hätten wir einen parameterlosen Konstruktor, hätte Powershell keinen Fehler gemeldet.

Daran ist prinzipiell auch nichts Verwerfliches, da wir in der Oberklasse über den Konstruktor das Setzen von Zuständen implementiert haben, die für unser Verhalten wichtg sind (Titel und Spielzeit). Auch eine CD sollte über diese Informationen verfügen. Der Konstruktor von CD muss deshalb diese Informationen an den Konstruktor der Oberklasse Item weiterreichen.

Dies erfolgt durch das Weiterleiten im Konstruktoraufruf:

class CD : Item

{

[string]$artist

[int]$numberOfTracks

CD([string]$a, [int]$n, [string]$t, [int]$p):base($t, $p){

$this.artist = $a

$this.numberOfTracks = $n

}

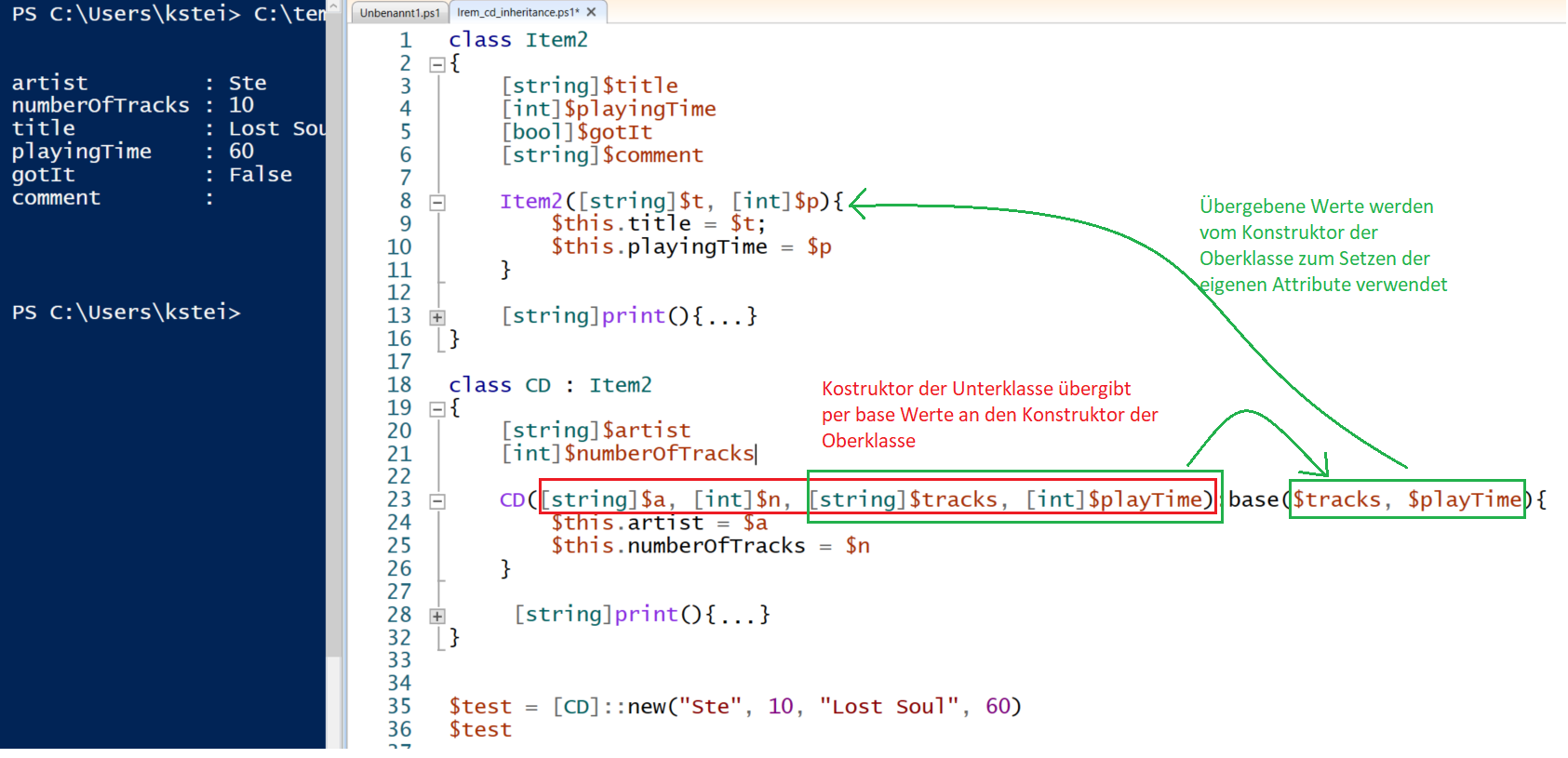
[string]print(){

return $this.title + ", " + $this.artist + ", " `

+ $this.numberOfTracks + ", " + $this.playingTime + " Minuten"

}

}

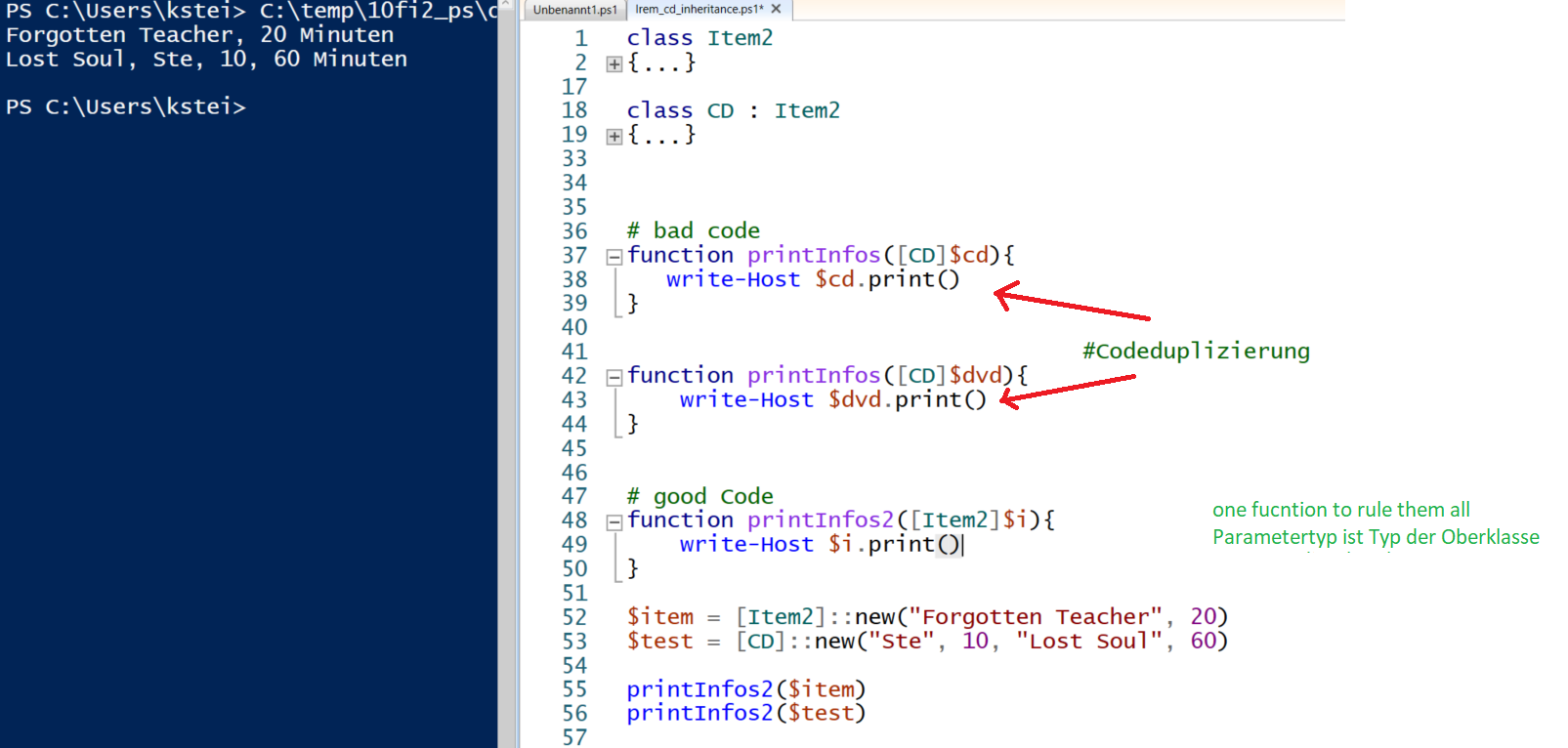


#### Vererbung als Vertrag

Durch die Vererbungsbeziehung hat ein Objekt der Unterklasse immer auch die Fähigkeiten und Methoden der Oberklasse. Das bedeutet, dass überall wo ein Objekt der Oberklasse erwartet wird, ein Objekt der Unterklasse verwendet warden kann.

Folgendes Beispiel mag dies verdeutlichen

Es gibt innerhlab ihres Skriptes die Funktion printInfos(), welches sich Informationen zu CDs und DVDs ausgeben möchte. Da dies zwei verschiedene Typen sind, nüsste die Funktion zweimal mit unterschiedlichen Paramtertypen geschrieben warden. Dies funktioniert nicht in Powershell und würde weiterhin zu unschöner Codeduplizierung führen. Nicht nur in Powershell würde man das Problem durch die Verwendung der Oberklasse als Parametertyp lösen.

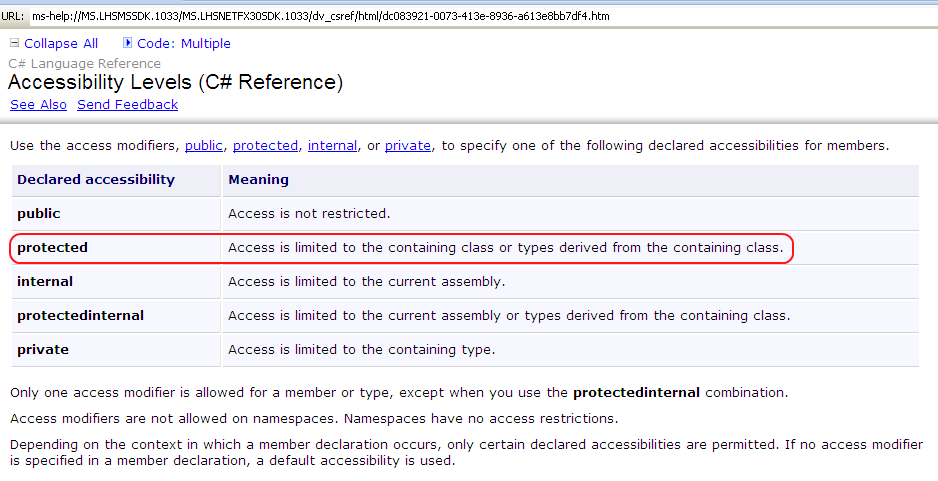


#### Zugriff auf Attribute und Methoden der Oberklasse

Durch die Vererbung wird zunächst der Zugriff auf öffentliche Methoden und Attribute gewährt. Private Attribute und Methoden sind durch das Prinzip der Kapselung weiter geschützt.

Will man den Unterklassen Zugriff auf diese Elemente gewähren, so müssen die Zugrifsmodifizierer der Oberklasse geändert werden.

In vielen Programmiersprachen (z.B. C#) stehen folgende Modifizierer zur Verfügung:



Leider kennt Powershell nicht diese feinen Unterschiede und lässt den Zugriff auf die Attribute der Oberklasse zu.

#### Überschreiben von Methoden der Oberklasse

Häufig ist es so, dass eine Methode der Oberklasse nicht exakt so implementiert ist, wie es sich die Unterklasse wünscht. In diesem Falle kann die Unterklasse die Methode der Oberklasse neu schreiben, d.h. man schreibt/implementiert exakt den gleichen Methodennamen in der Unterklasse

Die Methode print() der Klasse Item gibt zur Zeit den Titel sowie die Spielzeit aus.

class Item2

{

[string]$title

[int]$playingTime

[bool]$gotIt

[string]$comment

Item2([string]$t, [int]$p){

$this.title = $t;

$this.playingTime = $p

}

[string]print(){

return $this.title + ", " + $this.playingTime + " Minuten"

}

}

Dieses Verhalten kann eventuell nicht das sein, was Instanzen der Klasse CD möchten. Doch wie kann die Ausgabe für Objekte der Klasse CD geändert werden, ohne eine andere Methode benutzen zu müssen. Um es allgemeiner auszudrücken:

Wie kann eine Unterklasse ein anderes Verhalten als das der Oberklasse implementieren?

Die Lösung besteht im Neudefinieren der Methode in den Unterklassen.

class CD : Item2

{

[string]$artist

[int]$numberOfTracks

CD([string]$a, [int]$n, [string]$tracks, [int]$playTime):base($tracks, $playTime){

$this.artist = $a

$this.numberOfTracks = $n

}

[string]print(){

return $this.title + ", " + $this.artist + ", " `

+ $this.numberOfTracks + ", " + $this.playingTime + " Minuten"

}

}

Eine weitere Möglichkeit bestünde darin, die bereits vorhanden Implementierung der Oberklasse zu nutzen und mit eigenen Ergänzungen zu versehen.

Die Klasse DVD soll dies demonstrieren.

class DVD : Item2

{

[string]$director

DVD([string]$d, [string]$title, [int]$playTime):base($title, $playTime){

$this.director = $d

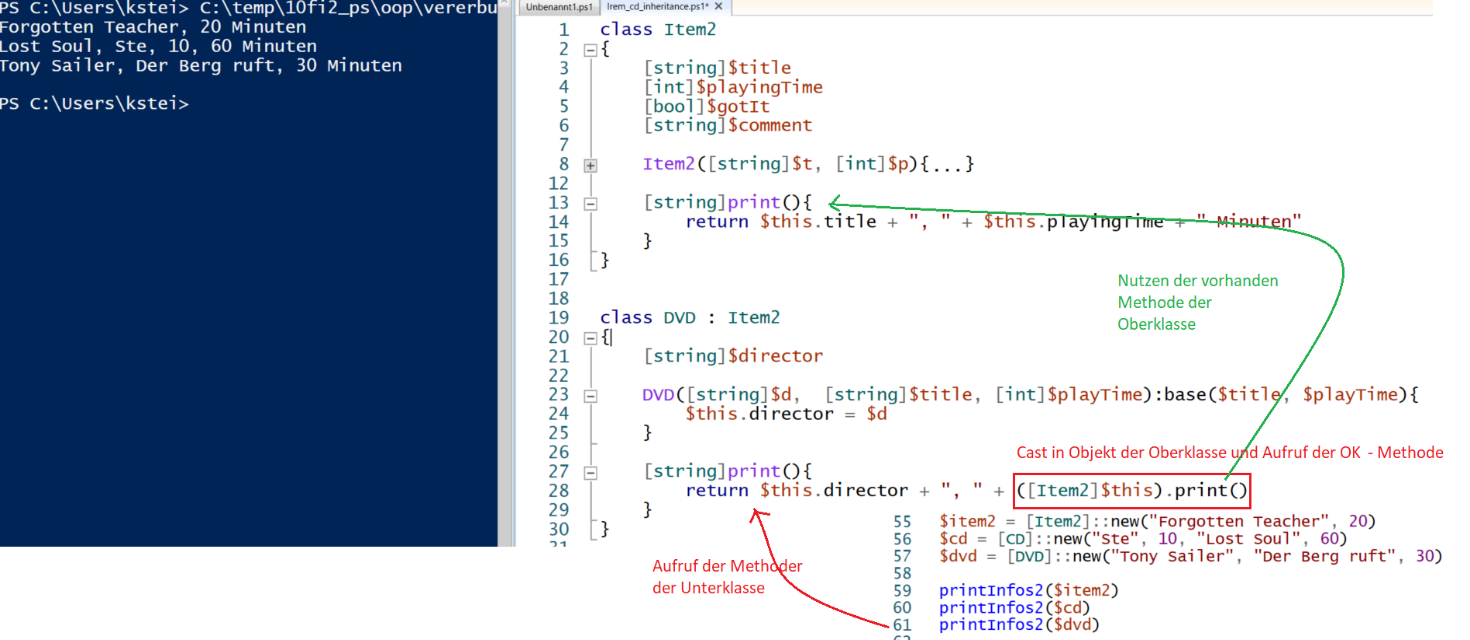
}

[string]print(){

return $this.director + ", " + ([Item2]$this).print()

}

}



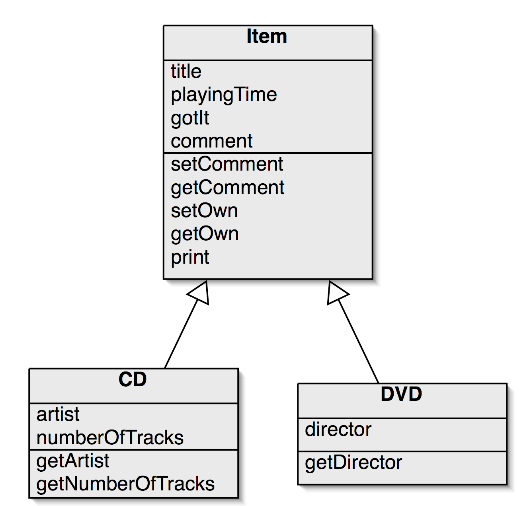
## Teil 3

## Der 3. Teil setzt sich mit folgenden Konzepten auseinander:

* Abstrakte Klasse / Interface

# Abstrakte Klasse

Die Implementierung des vorherigen Beispiels CD/DVD hatte einen großen Nachteil:



Es gibt eigentlich keine Instanzen von Item, sie waren aber jederzeit erzeugbar.

$item2 = [Item]::new("Forgotten Teacher", 20)

printInfos2($item2)

Das Konzept der Abstrakten Klasse kann dieser Problematik begegnen.

In vielen Programmiersprachen (C#, java) wird dies wie folgt umgestetzt.

* Definition der Klasse oder einer Methode als abstract
* Damit ist eine Instanziierung dieses Typs ausgeschlossen
* Klasse kann dennoch konkrete Implementierungen von Methoden sowie Attribute besitzen
* Methoden, von denen bekannt ist, dass Sie die Unterklassen überschreiben müssen, werden als abstrakt definiert. Sie bestehen lediglich aus der Methodensignatur.
* Damit müssen ableitende Unterklassen diese Methoden implementieren, ansonsten können keine Instanzen der Unterklassen gebildet werden.

Leider kann die bisherige Version von Powershell dieses Konzeot nicht umetzen, weil es das Sprachkonstrukt (**abstract)** nicht gibt.

**Man kann es aber simulieren.**

|  |
| --- |
| Aufgabe:  Überlegen Sie, was abstract eigentlich tatsächlich für eine Klasse bedeutet und überlegen sie, wie sie dies in Powershel umsetzen könntn.   * Objekt der Oberklasse darf nicht instanziierbar sein * Methode der Oberklasse darf evtl. nicht aufrufbar sein, muss aber eine Implementierung in der Unterklasser erzwingen   Lösung   * Konstruktor der Oberklasse muss prüfen, welcher Typ erzeugt wird und wirft u.Um eine Exception * Methoder der Oberklasse wirft eine Exception und sollte deshalb von Unterklasse überschrieben werden. |

class Item

{

[string]$title

[int]$playingTime

[bool]$gotIt

[string]$comment

Item([string]$t, [int]$p){

$type = $this.GetType()

if ($type -eq [Item])

{

throw("Class $type must be inherited")

}

$this.title = $t;

$this.playingTime = $p

}

[string]print(){

throw("Class method has to be overwritten")

}

}

class DVD : Item

{

[string]$director

DVD([string]$d, [string]$title, [int]$playTime):base($title, $playTime){

$this.director = $d

}

[string]print(){

return $this.director + ", " + $this.numberOfTracks + ", " + $this.playingTime + " Minuten"

}

}

class CD : Item

{

[string]$artist

[int]$numberOfTracks

CD([string]$a, [int]$n, [string]$title, [int]$playTime):base($title, $playTime){

$this.artist = $a

$this.numberOfTracks = $n

}

[string]keineEigenePrintMethodeMehr(){

return $this.title + ", " + $this.artist + ", " `

+ $this.numberOfTracks + ", " + $this.playingTime + " Minuten"

}

}

# good Code

function printInfos2([Item]$i){

write-Host $i.print()

}

$cd = [CD]::new("Ste", 10, "Lost Soul", 60)

$dvd = [DVD]::new("Tony Sailer", "Der Berg ruft", 30)

printInfos2($dvd) #geht

printInfos2($cd) #geht nicht

#darf nicht gehen

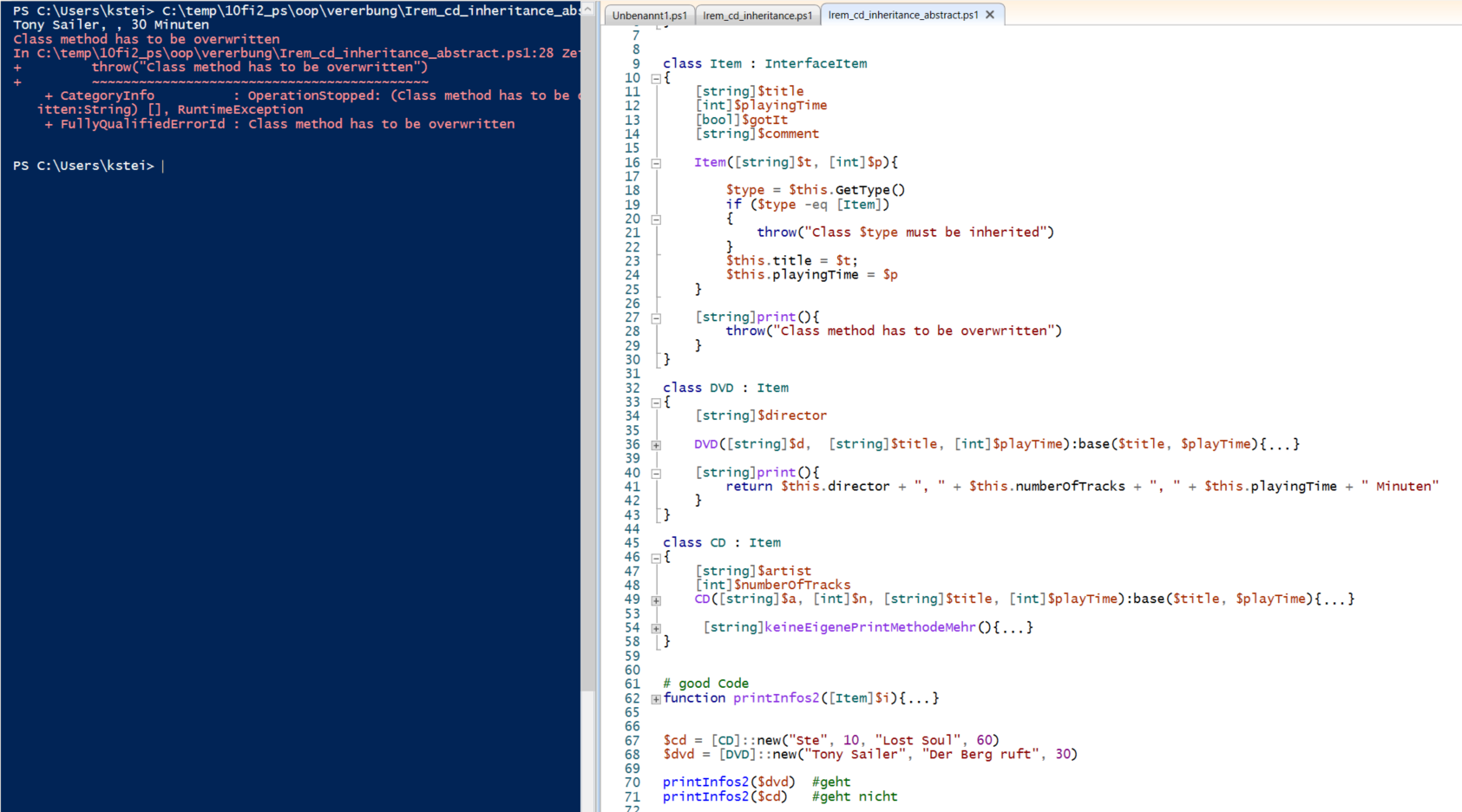
#$item2 = [Item]::new("Forgotten Teacher", 20)

#printInfos2($item2)

Falls versucht wird,

* Ein Objekt der Oberklasse bzw
* Eine nicht implementierte Methode der Oberklasse

aufzurufen, wird eine Ausnahme geworfen.



# Interface

Das Konzept des Interfaces führt die Idee der abstrakten Klasse weiter. Während die abstrakte Klasse durchaus in der Lage ist, eigene Attribute und Implementierungen von Methoden zu besitzen, reduziert sich das Interface im Prinzip lediglich auf die Deklaration von leeren Methodensignaturen. Diese werden dann von konkreten Klassen implementiert. Eine Implementierung mehrerer Inter-faces ist möglich (siehe Mehrfachvererbung).

Interfaces sind dann das geeignete Konzept, wenn es darum geht, die Schnittstellen von Klassen herauszuarbeiten, d.h. die Frage zu beantworten:

Wer kommuniziert mit mir über welche Methoden ?

Leider bietet Powershell noch nicht die Möglichkeit der Definition von Interfaces. Wie im Beispiel der abstrakten Klassen kann man es aber nachbauen. Man muss nur alle Methoden der Interface-Klasse mit einem Exception-Handling ausstatten.

class InterfaceItem{

[void]drive(){

throw("Class fdgdfg has to be overwritten")

}

}

class Item : InterfaceItem

{

[string]$title

[int]$playingTime

[bool]$gotIt

[string]$comment

Item([string]$t, [int]$p){

$type = $this.GetType()

if ($type -eq [Item])

{

throw("Class $type must be inherited")

}

$this.title = $t;

$this.playingTime = $p

}

[string]print(){

throw("Class method has to be overwritten")

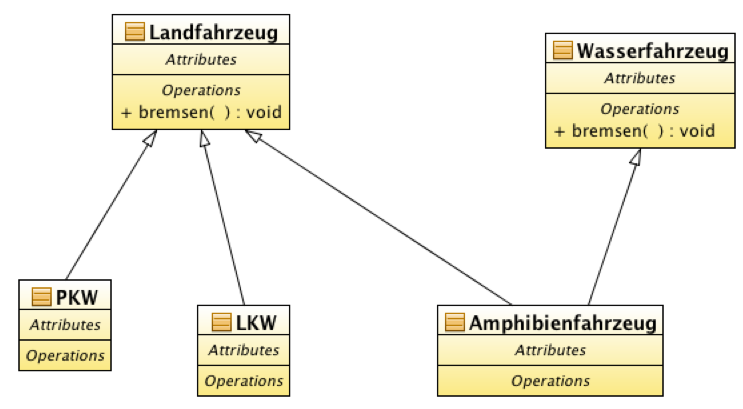
}

}

Nachteilig ist allerdings, dass manin Powershell dennoch nur von einer Klasse erben kann

# Mehrfachvererbung

Mehrfachvererbung erweitert die grundsätzliche Vererbung um die Fähigkeit, von mehreren Oberklassen gleichzeitig ableiten zu können. Damit ist aber auch bereits die Problematik vorgegeben.



Welche Methode bremsen() sollen Amphibienfahrzeuge ausführen ??

Aufgrund dieser Nebeneffekte haben sich die meisten Programmiersprachen vom Konzept der Mehrfachvererbung abgewendet, ohne jedoch die prinzipiellen Vorteile der Mehrfachvererbung mit Hilfe von Interfaces nicht zu implementieren.

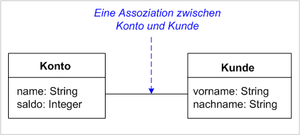
# Beziehungen zwischen Klassen

Während die Vererbung eine Typ-basierte-Beziehung darstellt, sind Beziehungen zwischen Klassen nicht nur auf diese Art herstellbar.

Objekte kommunizieren per Nachrichten mit anderen Objekten; häufig halten sie sich auch andere Obekte zur Erfüllung bestimmter Aufgaben.

## Assozation

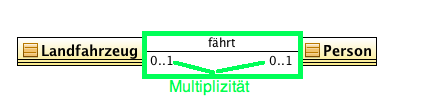
Unter Assoziation versteht man eine Beziehung zwischen zwei oder mehreren Typen. Untenstehende Abbildung zeigt dies in seiner einfachsten Form, einer Linie zwischen zwei Klassen. Sie drückt eine „Sich-Kennen“-Eigenschaft zwischen Objekten der jeweiligen Klasse aus.



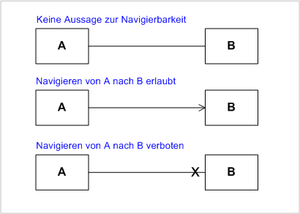
Die Linie kann durch verschiedene Elemente in ihrem Informationsinhalt erweitert werden.

### Multiplizität

Wie im Entity-Relationship-Modell kann die Assoziation um die Angabe erweitert werden, wieviele Objekte einer Seite jeweils mit einem Objekt der der anderen Seite in Beziehung stehen. Dies wird durch eine Zahl bzw. ein \* oder einer Min-Max-Notation zum Ausdruck gebracht.



### Navigierbarkeit

Eine Assoziation bildet eine Art Brücke zwischen zwei Typen: startet man bei der Instanz des einen beteiligten Typs kann man über eine Objektbeziehung zur Instanz des zweiten Typs navigieren. Die Navigierbarkeit von Assoziations-enden wird mit Hilfe eines > dargestellt und kann durch ein x eingeschränkt werden. Dabei unterscheidet man drei Arten, wie die Navigierbarkeit festgelegt werden kann:

* Keine Aussage (keine Pfeile)
* Erlaubte Navigation
* Nicht erlaubte Navigation

### Rollen

Gerade wenn es mehrere Assoziationen zwischen zwei Klassen gibt, kann der Informationsgehalt durch die Darstellung der Rolle erweitert werden. Die Rolle bezeichnet die Funktion, die ein Teilnehmer der Beziehung gegenüber der anderen Seite spielt. Sie wird an das jeweilige Assoziationsende geschrieben.

Im vorliegenden Beispiel gibt es zwei Beziehungen zwischen Bestellung und Adresse. Durch die unterschiedlichen Rollennamen kann man sie gut unterscheiden. Die beiden kleinen Dreiecke neben dem Assoziationsnamen unterstützen die Leserichtung.

**Adresse [ist] Rechnungsadresse für Bestellung**

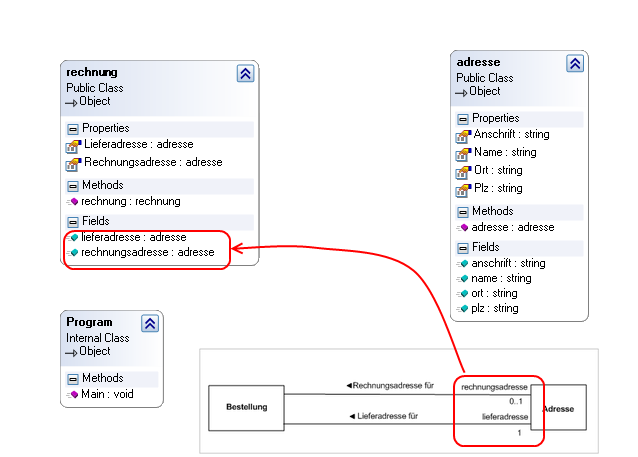
### Implementierung der Assoziation in einer Programmiersprache

Obenstehendes Beispiel von Adresse und Rechnung soll als Ausgangsgrundlage einer Implementierung dienen.

Folgende Grundüberlegungen können vorgenommen werden:

* Da keine Navigation angegeben ist, nehmen wir an, dass die Rechnung die Adresse kennt, nicht jedoch umgekehrt
* Ein Rechnungsobjekt hat 0 oder eine Rechnungs- bzw. Lieferadresse, d.h. sie muss jeweils eine Referenz auf maximal eines dieser beiden Adressen haben; dies kann durch eine entsprechende Referenzvariable erreicht werden.
* Die Rollennamen sind geeignete Namen für die Namen der Referenzvariablen.

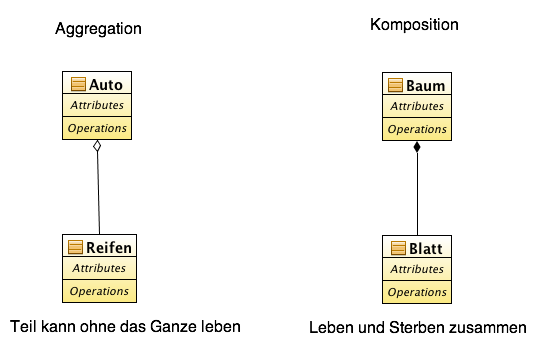
Aufgabe: Implementieren Sie untenstehende detaillierte Klassendiagrammansicht

****

## Aggregation/Komposition

Diese beiden Begriffe spiegeln eine besondere Art von Beziehungen zwischen Objekten wieder, nämlich sog. Teil-Ganzes-Beziehungen zwischen den Klassen. Es wird damit auch eine gewisse Abhängigkeit zwischen den Klassen ausgedrückt.

So ist das Ganze zur Erfüllung seiner Aufgaben von den Teilen abhängig (Aggregation), während zusätzlich dazu ein Teil ohne das Ganze nicht existieren kann (Komposition). In beiden Fällen kann das Ganze seine Aufgaben nur durch die Teile erfüllen, das Ganze wird damit erst durch seine Teile zu einem sinnvollen Objekt.



Aggregation und Komposition im Vergleich

Die Frage, ob eine Beziehung als Aggregation oder Komposition gestaltet werden soll, hängt häufig vom jeweiligen Problem ab und kann nicht immer gleich beantwortet werden.

Die Entscheidung für Aggregation und Komposition hat aber gewisse Konsequenzen für die Implementierung, insbesondere bei der Komposition:

* Das Ganze darf bei der Komposition die alleinige Referenz auf das Teil besitzen
* Deshalb muss das Ganze das Teil selbst erzeugen
* Das Ganze darf keine Referenzen auf das Teil an Klienten weitergeben, da sonst die Kontrolle verloren gehen kann.

<https://powershellexplained.com/2017-04-10-Powershell-exceptions-everything-you-ever-wanted-to-know/>

# Fehlerbehandlung

Siehe dazu auch:

<https://powershellexplained.com/2017-04-10-Powershell-exceptions-everything-you-ever-wanted-to-know/>

Fehlerbehandlung ist ein elementarer Bestandteil einer jeden Programmierung und umfasst einen großen Bestandteil des Quellcodes. Das Managen von Fehlern macht den Code insgesamt robister und das laufende Programm ist damit stabiler. Powershell kennt verschiedene Arten, wie man Fehler verwalen kann.

## What-if

Um zu sehen, welche Effekte ein Skript hat, kann man es mit dem sog. What-if-Paramter versehen.

# What exactly would happen if Stop-Process

# ended all processes beginning with "c"?

Stop-Process -Name c\* -WhatIf

PS C:\> Stop-Process -Name c\* -WhatIf

WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "conhost (1012)".

WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "conhost (2000)".

WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "csrss (368)".

WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "csrss (428)".

What-IF kann auch in eigene Funktionen integriert werden.

function MapDrive([string]$driveletter, [string]$target, [switch]$whatif)

{

If ($whatif)

{

Write-Host "WhatIf: creation of a network drive " +

"with the letter ${driveletter}: at destination $target"

}

Else

{

New-PSDrive $driveletter FileSystem $target

}

}

MapDrive k \\127.0.0.1\C$ -whatif

WhatIf: creation of a network drive + with the letter k: at destination \\127.0.0.1\C$

## Confirm

Mit Hilfe des -confirm - Parameters wird vor der Ausführung eines Befehls eine Sicherheitsabfrage formuliert.

Stop-Service a\* -Confirm

Bestätigung

Möchten Sie diese Aktion wirklich ausführen?

Ausführen des Vorgangs "Stop-Service" für das Ziel "Anwendungserfahrung

(AeLookupSvc)".

[J] Ja [A] Ja, alle [N] Nein [K] Nein, keine [H] Anhalten [?] Hilfe

(Standard ist "J"):n

Bestätigung

Möchten Sie diese Aktion wirklich ausführen?

Ausführen des Vorgangs "Stop-Service" für das Ziel "Gatewaydienst auf

Anwendungsebene (ALG)".

[J] Ja [A] Ja, alle [N] Nein [K] Nein, keine [H] Anhalten [?] Hilfe

(Standard ist "J"):

Da manche Befehle „kritischer“ als andere Befehle sind, haben die Schöpfer der **Powershell** verschiedene Confirm-Level (Low, Medium, High, None) eingebaut und diese an bestimmte Befehle gebunden. So würde die Bestätigungsabfrage beim Löschen einer User-Mail per exchange-cmdlet auch dann eine Sicherheitsabfrage hervorrufen, wenn der -confirm - Parameter nicht übergeben wird. Dies kann über die Variable **$ConfirmPreference** gesteuert werden. Wenn diese auf „Low“ gesetzt wird, wird immer einer Bestätigung verlangt, bei „None“ findet keine Bestätigung statt.

Calculator may be started and stopped without being called

# into question because Stop-Process is in the Medium category:

Calc

Stop-Process -Name calc

# If the default setting is changed from High to Low,

# PowerShell will automatically question every action:

$ConfirmPreference = "Low"

calc

Stop-Process -Name calc

## ErrorAction

Auf einen Fehler kann Powershell mit verschiedenen Verhaltensweisen reagieren.

* SilentlyContinue: Einfach weitermachen
* Continue: Fehlermeldung anzeigen aber weitermachen
* Stop: Ausführung des Skriptes anhalten
* Inquire: Nachfragen, wie es sich verhalten soll.

Das Verhaltensweisen können mit Hilfe des Parameter -ErrorAction bzw. der globalen Variable $ErrorActionPreference gesteuert werden.

Del "nosuchthing"; Write-Host "Done!"

Del "nosuchthing" -ErrorAction "SilentlyContinue"; Write-Host "Done!"

$script:ErrorActionPreference = "Stop" #globales Anhalten bei Fehlern

## Erkennen und Behandeln von Fehlern

Häufig will man auf Fehler auch reagieren können. Eine Möglichkeit ist es, den Fehlerstatus eines Programms auszuwerten. Dieser kann über die globale Variable **$?** abgefragt werden. Wenn ein Fehler aufgetreten ist, hat diese Variable den Wert **True**.

Del "nosuchthing" -ErrorAction "SilentlyContinue"

If (!$?) { "Didn't work!"; break }; "Everything's okay!"

Del "nosuchthing" -ErrorAction "SilentlyContinue"

If (!$?) { "Error: $($error[0])"; break }; "Everything's okay!"

Error: Cannot find path "u:\nosuchthing" because it does not exist.

Wenn Sie wissen wollen, welcher Fehler tatsächlich aufgetreten ist, kann mit Hilfe der $Error-Variable den tatsächlichen Fehler herausfinden. **$Errror** ist ein Array, der an der Indexstelle 0 den jeweils aktuellsten Fehler hat

## 12.5. Benutzen von Traps

Als Alternative können sog. „**Traps**“ verwendet werden. Wenn bekannt ist, dass ein bestimmter Befehl eventuell nicht zur Laufzeit korrekt funktioniert, dann kann man dies mit Hilfe eines Traps definieren, was im Fehlerfall passieren soll.

Trap { "A dreadful error has occurred!"} 1/$null

A dreadful error has occurred!

Attempted to divide by zero.

At line:1 char:53

+ Trap { "A dreadful error has occurred!"} 1/$ <<<< null

* Traps Require Unhandled Exceptions

Traps funktionieren nur mit Fehlern, die nicht von anderer Stelle „gehandelt“ werden. Das untere Statement gibt deshalb keine Fehlemeldung aus, der der Compiler den Teilungsversuch der beiden Konstanten selbst behandln kann. Ebenfalls werden Commandlets häufig eine eigene Fehlerbehandlung haben. Diese kann man durch Setzen der $ErrorAction - Variable beeinflussen.

Trap { "A dreadful error has occurred!" } Del "nosuchthing"

del : Der Pfad "C:\Users\steinam\nocsuchthing" kann nicht gefunden werden, da er nicht vorhan..

In Zeile:1 Zeichen:42

+ Trap { "A dreadful error has occurred!"} del "nocsuchthing"

+ ~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound: (C:\Users\steinam\nocsuchthing:String) [Remove-...

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.RemoveItemCommand

Trap { "A dreadful error has occurred!" } `

Del "nosuchthing" -ErrorAction "Stop"

A dreadful error has occurred!

Del : Der Pfad "C:\Users\steinam\nosuchthing" kann nicht gefun...

## 12.6. Exceptions verstehen

Fehler werden in modernen Programmiersprachen häufig mit dem neutralen Wort **Ausnahmen** bezeichnet. Beim Auftreten eines Fehlers wird eine Ausnahme ausgelößt, die von einer Stelle im Quellcode behandelt werden muss; ansonsten kommt es letztlich zu einer roten Fehlermeldung auf der Konsole.

Ausnahmen sind letztlich spezielle Fehlerklassen des .NET-Frameworks; für bestimmte Fehler gibt es jeweils spezielle Fehlerklassen.

In Abhängigkeit von bestimmten Fehler kann ein Skript nun unterschiedlich reagieren. Dies wird mit Hilfe des **try – catch – finally** - Konstruktes möglich.

### Try, Catch, Finally

Try, Catch und Finally wurde in der PowerShell 2.0 implementiert und verhält sich ähnlich, wie das gleich benannte Konstrukt in C#. Damit kann eine sehr elegante Fehlerbehandlung in der Powershell realisiert werden. Die Syntax lautet wie folgt:

**Try**                                 Try Schlüsselwort

**{**

**<StatementList>**                 Anweisungsblock für Try

**}**

**Catch**                              Catch Schlüsselwort

**[[<Error-Type>][,<ErrorType>]\*]**     Optionale Fehlertypen

**{**

**<StatementList>**                     Anweisungsblock für Catch

**}**

**Finally**                             Finally Schlüsselwort

**{**

**<StatementList>**                     Anweisungsblock für Finally

**}**

Der Catch-Block ist optional, es kann auch mehrere Catch-Blöcke geben. Der Finally-Block ist ebenfalls optional, es kann nur einen geben. Die Fehlerbehandlung funktioniert wie folgt:

* Im Try-Block werden die Anweisungen platziert, welche einen Fehler (Terminating Error) verursachen können (z.B. Zugriff auf eine Webseite – die Internetverbindung kann unterbrochen sein, der Server kann offline sein, etc.)
* Im Catch-Block erfolgt die Reaktion auf einen möglichen Fehler (im Anweisungsblock). Der Catch-Block wird auch nur dann ausgeführt, wenn ein Fehler aufgetreten ist. Wird im Catch-Block kein Fehlertyp aufgeführt, wird er bei allen Fehlern ausgeführt, die im Try-Block aufgetreten sind. Gibt es mehrere Catch-Blöcke mit spezifizierten Fehlertypen, wird derjenige ausgeführt, der als erster den aufgetretenen Fehlertyp abdeckt. Deshalb ist es bei mehreren Catch-Blöcken sinnvoll, die am meisten spezifischen zuerst im Code zu platzieren.
* Der Finally-Block wird immer ausgeführt, unabhängig davon, ob ein Fehler aufgetreten ist oder nicht (es sei denn, Ctrl + C wird zwischendurch gedrückt). Der Finally-Block wird meist dazu verwendet, nicht mehr benötigte Objekte und Variablen zu entfernen.

***Beispiel:***

|  |  |
| --- | --- |
| 1.  Try {  2.  $Winword = New-Object -ComObject Word.Application  3.  }  4.  Catch {  5.  Write-host "No Winword installed"  6.  }  7.  Finally {  8.  $Winword = $Null  9.  }  10.  11. Try {  12. $MyApp = New-Object -ComObject MyApp.Application  13. }  14. Catch {  15. Write-host "MyApp is not installed"  16. }  17. Finally {  18. $MyApp = $Null  19. }  20. |  |

Das erste Try-Konstrukt versucht ein neues COM-Objekt aus Winword zu erstellen. Da aber Winword auf dem Computer installiert ist und kein sonstiger Fehler auftritt, wird der Catch-Block gar nicht ausgeführt, dafür aber Finally. In dem zweiten Konstrukt wird das gleiche mit einer imaginären Applikation „MyApp“ versucht. Da sie aber nicht existiert, wird eine Exception erzeugt und somit auch der Catch-Block ausgeführt (Ausgabe in der letzten Zeile).

Natürlich ist das hier ein einfaches Beispiel. In Wirklichkeit würde der Try-Block noch mehr mit dem COM Objekt machen…

## 12.8. Aufgaben

**Aufgabe1**:  
Schreiben Sie eine Funktion, die als Parameter eine URL-Ressource erhält. Diese Ressource soll dann heruntergeladen und gepsiechert werdn.

Bsp: http://www.steinam.de/download/schulaufgabe.docx

Überlegen Sie sich, welche Fehler dabei auftreten können

Implementieren Sie die Funktion mit Hilfe von try/catch

**Aufgabe 2**:

Finden sie den Unterschied zwischen den sog

**Terminierenden bzw. Nicht terminierenden Fehlern**

**Aufgabe 3**:

Schreiben Sie eine Funktion, die zwei Zahlen miteinander dividiert. Wenn versucht wird durch die Zahl 0 zu teilen, soll eine Hinweis erscheinen: „Sie versuchen durch 0 zu teilen

**Aufgabe 4**:

Eine Funktion erhält eine Liste von Dateinamen (incl. Pfadangabe), sowie ein Ziellaufwerk. Die Funk-tion soll nun die durchgehen und die Dateien in das entsprechende Ziellaufwerk mit exakt gleicher Ordnerstruktur kopieren kopieren.

Überlegen Sie sich mögliche Fehlerquellen und fangen Sie diese per try/catch-Statement ab.

**Aufgabe 5**:

Schreiben Sie eine Funktion, die versucht, per remote-Zugriff das Betriebssystem eines anderen Rechners abzufragen

Die entsprechend abzusichernde Anwesiung sieht letztendlich wie folgt aus:

$OS = Get-WmiObject -ComputerName RemoteServer -Class Win32\_OperatingSystem -ErrorAction Stop

Überlegen Sie sich, welche mögliche Fehlerquellen ein Zugriff haben könne. Schreiben Sie für jede diese Ursachen einen entsprechenden catch-Block. Am Ende sollte ein allgemeiner catch-Block mögliche Restausnahmen abfangen.

Aufgabe 6:

Arbeiten Sie die Codebeispiele auf folgender englischsprachiger Webseite durch

<https://www.gngrninja.com/script-ninja/2016/6/5/powershell-getting-started-part-11-error-handling>