EINFÜHRUNG IN DIE PROGRAMMIERUNG

OBJEKTORIENTIERTE PROGRAMMIERUNG

DHBW MANNHEIM
WIRTSCHAFTSINFORMATIK (DATA SCIENCE)

Markus Menth
Martin Gropp

GRUNDLAGEN

OBJEKTE, ATTRIBUTE UND METHODEN

Was ist eigentlich ein Objekt?

Objekt = Zustand & Aktionen

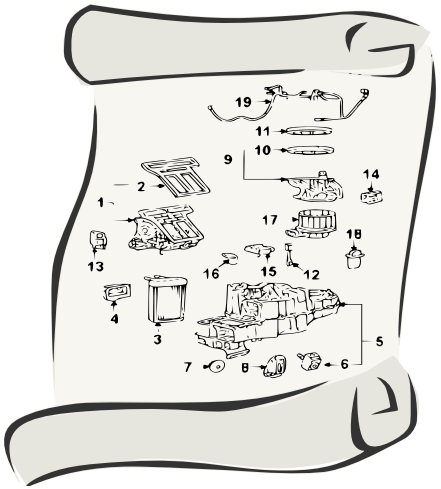
…?

Beispiel: ein Fahrzeug

- Zum Zustand des Fahrzeugs gehören beispielsweise seine *Position*, *Tankfüllstand*, *Farbe*, *Leistung*, …
  Der Zustand wird beschrieben durch **Attribute** — Variablen eines Objektes.
- Eine Aktion wie z.B. *fahren* oder *tanken* kann diesen Zustand verändern.
  Die Aktionen werden beschrieben durch **Methoden** — Funktionen eines Objektes.

KLASSEN

Die Beschreibung, welche Attribute und Methoden Objekte haben sollen, erfolgt durch **Klassen**, die sozusagen den Bauplan für ein Objekt darstellen.

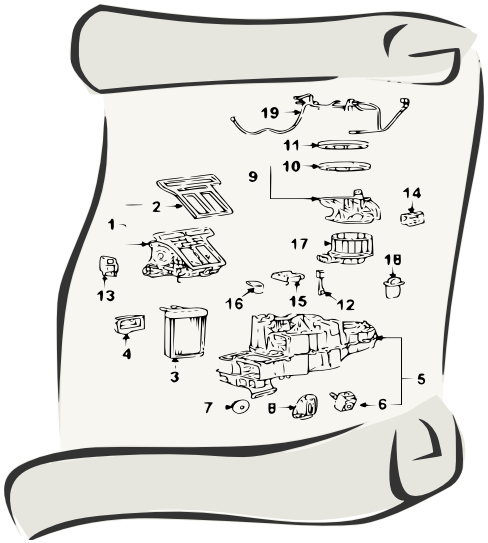



```
class Fahrzeug:
    # Attribute
    # Methoden
```

OBJEKTE

Um **Objekte** ("Instanzen" einer Klasse) anzulegen, verwendet man dieselbe Syntax wie bei Funktionsaufrufen:

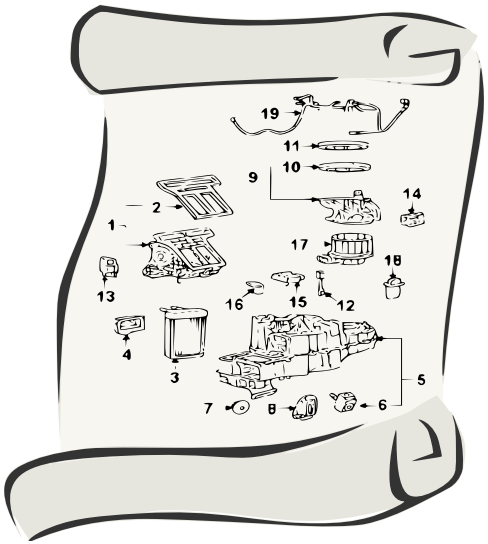
```
f = Fahrzeug()
```



→ `Fahrzeug()`

OBJEKTE

Natürlich kann ein Bauplan wiederverwendet werden.



→

`Fahrzeug()`

OBJEKTE

```
f1 = Fahrzeug()
f2 = Fahrzeug()
f3 = Fahrzeug()

f = [Fahrzeug(), Fahrzeug(), Fahrzeug()]
```

KLASSEN

DIE EINZELTEILE: CLASS

```
class Fahrzeug:
    def __init__(self) -> None:
        self.x = 0
        self.y = 0

    def fahren(self, x: int, y: int) -> None:
        self.x = x
        self.y = y
```

Klassen werden angelegt mit dem Schlüsselwort `class`.

Klassen sind **Typen**, der Typ eines Objektes ist seine Klasse.

KLASSEN

DIE EINZELTEILE: METHODEN

```python
class Fahrzeug:
    def __init__(self) -> None:
        self.x = 0
        self.y = 0

    def fahren(self, x: int, y: int) -> None:
        self.x = x
        self.y = y
```

Methoden sind spezielle Funktionen innerhalb einer Klasse.
Sie haben einen besonderen Parameter `self`: das Objekt selbst.

⚠ Das ist in den meisten Sprachen anders!

KLASSEN

DIE EINZELTEILE: __INIT__

```
class Hallo:
    def __init__(self) -> None:
        print('Hallo!')
```

Die Methode `__init__` ist eine besondere Methode, die beim Erstellen eines Objektes ausgeführt wird.

```
>>> h = Hallo()
Hallo!
```

KLASSEN

DIE EINZELTEILE: ATTRIBUTE

```
class Fahrzeug:
    def __init__(self) -> None:
        self.x = 0
        self.y = 0

    def fahren(self, x: int, y: int) -> None:
        self.x = x
        self.y = y
```

In Python wird `__init__` verwendet, um **Attribute** anzulegen, hier `self.x` und `self.y`.

ZUGRIFF AUF ATTRIBUTE UND METHODEN

```python
class Fahrzeug:
    def __init__(self) -> None:
        self.x = 0
        self.y = 0

    def fahren(self, x: int, y: int) -> None:
        self.hupen()
        self.x = x
        self.y = y

    def hupen(self) -> None:
        print('TRÖÖÖÖÖT')
```

Innerhalb von Methoden einer Klasse kann mit `self.`⟨Name⟩ auf Attribute und andere Methoden zugegriffen werden.

ZUGRIFF AUF ATTRIBUTE UND METHODEN

Auf die Attribute und Methoden eines Objektes `f` kann von außen mit `f.⟨Name⟩` zugegriffen werden:

```
f = Fahrzeug()

x = f.x
y = f.y

f.fahren(x+1, y+1)
```

`self`?

Beim Aufrufen von Methoden wird für `self` kein Wert übergeben.
`self` verweist immer auf das Objekt.

ARBEITEN MIT OBJEKTEN

Änderungen an einem Objekt wirken sich normalerweise nicht auf andere Objekte aus.

```
f1 = Fahrzeug()
f2 = Fahrzeug()

f2.fahren(100, 250)

print('Fahrzeug 1:', f1.x, f1.y)
print('Fahrzeug 2:', f2.x, f2.y)
```

```
Fahrzeug 1: 0 0
Fahrzeug 2: 100 250
```

INIT

`__init__` kann, wie andere Funktionen auch, weitere Parameter haben.

```
class Fahrzeug:
    def __init__(self, x: int = 0, y: int = 0) -> None:
        self.x = x
        self.y = y
```

Werte für diese Parameter werden beim Anlegen des Objektes übergeben:

```
f = Fahrzeug(30, 80)
```

KAPSELUNG

KAPSELUNG

Bisher kann jedes Attribut auch von außerhalb beliebig verändert werden.

```
f = Fahrzeug()
f.tankstand = 1000000
```

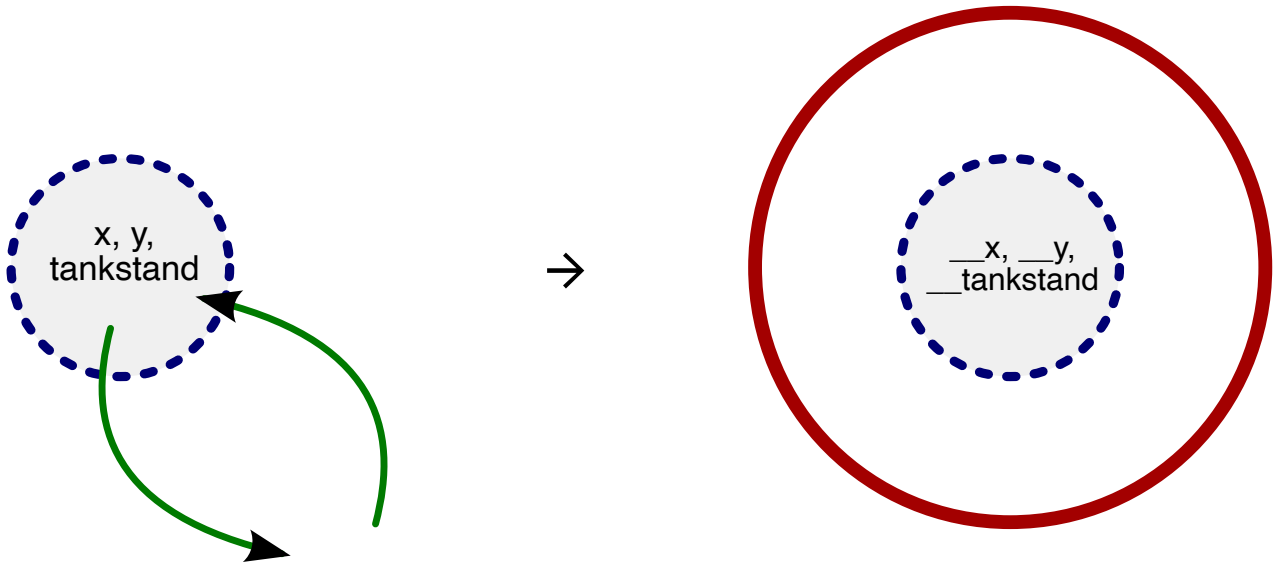
…so viel passt aber gar nicht in den Tank!

Kapselung
Attribute und Methoden werden als **privat** markiert, indem man einen Namen wählt, der mit zwei Unterstrichen __ beginnt.
Der Zugriff ist dann nur von innerhalb der Klasse "möglich".

KAPSELUNG

```
class Fahrzeug:
    def __init__(self) -> None:
        self.__x = 0
        self.__y = 0
        self.__tankstand = 50
```

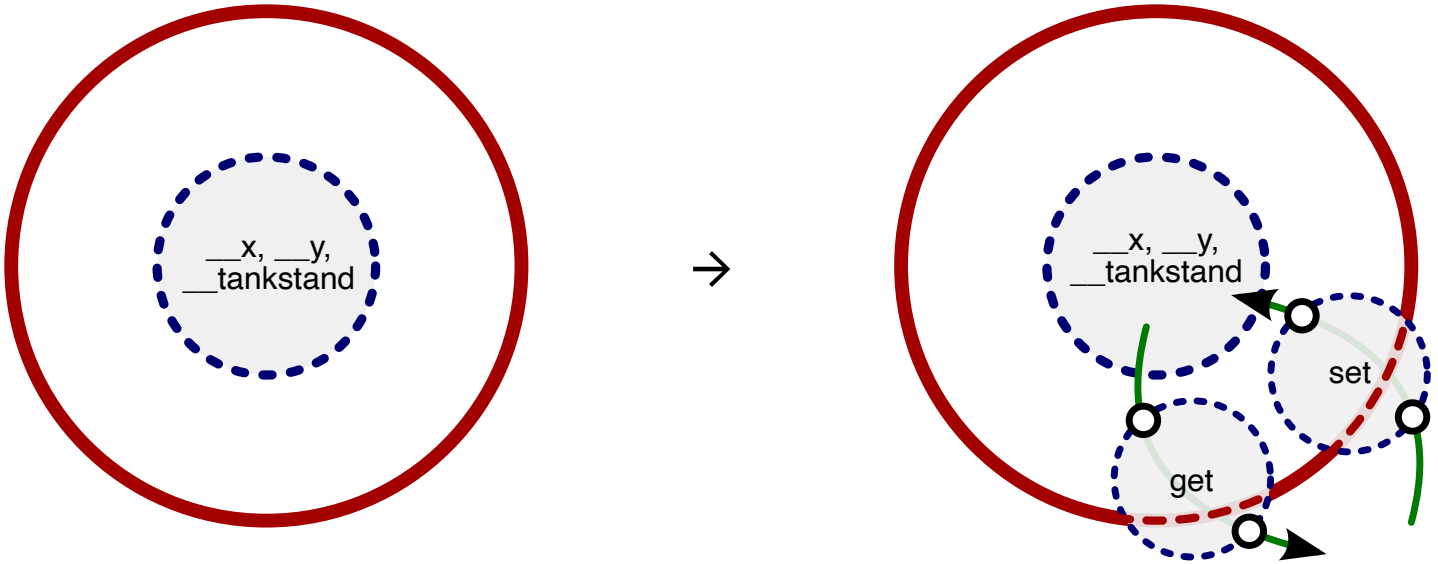
KAPSELUNG



KAPSELUNG

Jetzt kann das Fahrzeug aber gar nicht mehr tanken…

get/set-Methoden dienen dem kontrollierten Zugriff auf gekapselte Attribute von außen und sind in der Regel öffentlich.



KAPSELUNG

```
class Fahrzeug:
    def __init__(self) -> None:
        self.__tankstand = 50

    def get_tankstand(self) -> int:
        return self.__tankstand

    def set_tankstand(self, wert: int) -> None:
        if 0 <= wert <= 50:
            self.__tankstand = wert
```

Weiterführend: Properties 🔬

UNDER THE HOOD 🔬

Python unterscheidet sich bei der Behandlung von "privaten" Attributen und Methoden von vielen anderen Programmiersprachen.

- Namen, die mit einem Unterstrich `_` beginnen, werden als *weak private* betrachtet. Abgesehen von kleinen Einschränkungen (`import *`) *könnte* man eigentlich normal auf sie zugreifen (sollte das aber natürlich nicht).
- Auf Namen, die mit zwei Unterstrichen `__` beginnen (aber nicht enden), kann man nicht normal zugreifen. Tatsächlich verändert Python aber nur ihre Namen, aus `__x` in der Klasse `Fahrzeug` wird intern `_Fahrzeug__x`. Innerhalb der Klasse wird das automatisch übersetzt, von außen (und aus abgeleiteten Klassen) sieht es so aus, als ob `__x` nicht existieren würde.
  ⚠ Kein Schutz!
- Namen, die mit zwei Unterstrichen beginnen *und enden*, haben eine besondere Bedeutung. Hierzu gehört die `__init__`-Methode.

KLASSENVARIABLEN

KLASSENVARIABLEN

Beispiel: Seriennummern
Die Fahrzeuge sollen beim Bau der Reihe nach mit 1, 2, 3, … durchnummeriert werden.

Problem:
Jedes Objekt hat seine eigenen Attribute. Änderungen in einem Objekt wirken sich nicht auf andere aus. So kann ein Objekt nicht wissen, welche Seriennummer als nächstes vergeben werden muss.

KLASSENVARIABLEN

Idee:
Wenn sich alle Instanzen ein gemeinsames Attribut teilen würden, könnte man darin eine "Strichliste" führen, wie viele Fahrzeuge bereits gebaut wurden. Damit wüsste man beim Anlegen eines neuen Objekts, wie viele andere schon erstellt wurden, und könnte ihm seine Seriennummer zuweisen.

KLASSENVARIABLEN

Klassenvariablen gehören zur Klasse, nicht zu einem Objekt.
Änderungen sind in allen Instanzen sichtbar!

Bei Klassenvariablen verwendet man nicht `self` sondern den Klassennamen.

```
class Fahrzeug:
    __anzahl_gebaut = 0

    def __init__(self) -> None:
        Fahrzeug.__anzahl_gebaut += 1
        self.seriennummer = Fahrzeug.__anzahl_gebaut
```

```
f1 = Fahrzeug()
f2 = Fahrzeug()

print(f1.seriennummer, f2.seriennummer)
```

Klassenvariablen werden auch **statische Attribute** genannt.

KLASSEN- UND STATISCHE METHODEN 

Analog zu Klassenvariablen ist es auch möglich, Klassenmethoden zu definieren, die ohne ein Objekt aufgerufen werden können.
Sie werden mit `@classmethod` gekennzeichnet und bekommen statt einer Instanz `self` die Klasse `cls` übergeben.

```
class C:
    @classmethod
    def foo(cls, arg: int) -> None:
        print(arg)

C.foo(10)
```

KLASSEN- UND STATISCHE METHODEN 🔬

Bei Statischen Methoden fehlt der spezielle Parameter `cls`.
Sie werden mit `@staticmethod` gekennzeichnet.

```
class C:
    @staticmethod
    def foo(arg: int) -> None:
        print(arg)

C.foo(10)
```

WIEDERHOLUNG

BEGRIFFE

- Typ
- Klasse
- Objekt
- Attribut
- Methode
- `__init__`
- Kapselung: privat, öffentlich, get/set-Methoden
- Klassenvariable, statisches Attribut

VERERBUNG

VERERBUNG

Die Fahrzeugtypen PKW und LKW haben viele Gemeinsamkeiten. Diese Gemeinsamkeiten muss man nicht in jeder der Klassen einzeln beschreiben, sondern man kann sie einmal in der gemeinsamen Oberklasse `Fahrzeug` implementieren.

Die abgeleiteten Klassen `PKW` und `LKW` können dann die Funktionalität von der Klasse `Fahrzeug` erben.

```
class PKW(Fahrzeug):
    # ...
```

```
class LKW(Fahrzeug):
    # ...
```

VERERBUNG

Die Elternklasse wird in Klammern hinter den Klassennamen geschrieben.

```python
class Fahrzeug:
    def __init__(self) -> None:
        self.x = 0
        self.y = 0

    def fahren(self, x: int, y: int) -> None:
        print('brrrrummm')
        self.x = x
        self.y = y
```

```python
class PKW(Fahrzeug):
    def einsteigen(self) -> None:
        print('Herzlich willkommen!')
```

```python
class LKW(Fahrzeug):
    def beladen(self) -> None:
        print('Ladung gesichert!')
```

PKW und LKW erben alle Methoden von `Fahrzeug` und können individuell erweitert werden.

VERERBUNG

```
pkw = PKW()
pkw.einsteigen()
pkw.fahren(10, 20)
```

```
Herzlich willkommen!
brrrrummm
```

```
lkw = LKW()
lkw.beladen()
lkw.fahren(100, 100)
```

```
Ladung gesichert!
brrrrummm
```

ÜBERSCHREIBEN VON METHODEN

```
brrrrummm
```

…das ist eigentlich ein bisschen zu leise für einen LKW…

```
class LKW(Fahrzeug):
    def fahren(self, x: int, y: int) -> None:
        print("BBBBRRRRRRRRRRRRRRRRRUUUUUUUUUUUUUUUMMMMMMMMMMM")
        self.x = x
        self.y = y
```

Methoden der Oberklasse können **überschrieben** werden, indem man in der abgeleiteten Klasse eine Methode mit gleichem Namen anlegt.

ÜBERSCHREIBEN VON METHODEN

```
lkw = LKW()
lkw.fahren(100, 100)
```

BBBBRRRRRRRRRRRRRRRRRUU

UUUUUUUUUUUUMMMMMMMMMMM

ÜBERSCHREIBEN VON METHODEN: SUPER

Falls man die überschriebene Methode der Oberklasse noch braucht, kann man innerhalb der abgeleiteten Klasse mit dem Schlüsselwort `super` darauf zugreifen.

Das ist ganz besonders wichtig für die `__init__`-Methode!

```
class LKW(Fahrzeug):
    def __init__(self) -> None:
        super().__init__()
        self.ladung = 0
```

⚠ Ohne den *super*-Aufruf würde die `__init__`-Methode von `Fahrzeug` nicht aufgerufen und das Objekt hätte keine Attribute `x` und `y` mehr!

ÜBERSCHREIBEN VON METHODEN: SUPER

Natürlich kann man im *super*-Aufruf auch Parameter übergeben.

```
class Fahrzeug:
  def __init__(self, x: int, y: int, klasse: str) -> None:
    self.x = x
    self.y = y
    self.klasse = klasse
```

```
class LKW(Fahrzeug):
    def __init__(self, x: int, y: int) -> None:
        super().__init__(x, y, 'C')
```

KAPSELUNG II

Im Gegensatz zu anderen Programmiersprachen hat Python keinen Mechanismus, um bestimmte Attribute und Methoden nur für abgeleitete Klassen sichtbar zu machen ("protected").

- x: öffentlich, auch von außen zugreifbar
- _x: sollte man nur innerhalb der Klasse selbst und aus abgeleiteten Klassen verwenden; IDEs warnen normalerweise, wenn man versucht, von außen zuzugreifen.
- __x: kann man (ohne Tricks) nur in der Klasse selbst verwenden (auch nicht in abgeleiteten Klassen)

WIEDERHOLUNG

BEGRIFFE

- Vererbung
- Oberklasse, Elternklasse, Basisklasse
- abgeleitete Klasse
- Überschreiben
- super

STRING-REPRÄSENTATIONEN

STRING-REPRÄSENTATION

Was passiert eigentlich, wenn man versucht, ein Objekt mit `print` auszugeben?

```
class Fahrzeug:
    def __init__(self) -> None:
        self.x = 0
        self.y = 0

f = Fahrzeug()
print(f)
```

```
<__main__.Fahrzeug object at 0x7f9f36380040>
```

__STR__

Intern wird, genau wie für `str(f)`, die Methode `__str__` aufgerufen und `f` damit als String dargestellt.

Für eine eigene String-Repräsentation kann man `__str__` überschreiben.

```
class Fahrzeug:
    def __init__(self):
        self.x = 0
        self.y = 0

    def __str__(self) -> str:
        return f'Fahrzeug(x={self.x}, y={self.y})'
```

```
Fahrzeug(x=0, y=0)
```

AUFGABE

AUFGABE

Implement a class `Person` with standard attributes such as `name`, `date_of_birth`, etc.

Use the classes datetime and timedelta from the `datetime` module: `from datetime import datetime, timedelta`.

- Add a constructor that initializes the individual attributes.
- Add a method that returns the current age of a `Person` object as a `timedelta` object.
- Add a method `__str__` that returns a summary of the person as a string (cf. this documentation).
- Add a functionality that counts the number of `Person` objects that have been created.

AUFGABE

Implement an address book that can store `Person` instances.

- Implement methods to add a person.
- Add three different Person `objects` to the address book.
- Implement a search functionality that searches all attributes for a user-supplied substring (e.g., `ab.search("0621")` or `ab.search("Peter")`).
- Implement a delete functionality.

AUFGABE

Implement a class `Customer` that inherits from `Person`.

- Add an attribute `customer_id`.
- Implement a constructor that accepts all parameters of the super constructor plus the `customer_id` and invoke the super constructor.
- Overwrite the function that prints a summary of the person on the console and additionally print the customer id.

Implement a class `Employee` that inherits from `Person`

- Similar to `Customer`, add an attribute `username`.

Store instances of this type in the address book.

DATENKLASSEN

DATENKLASSEN

Datenklassen sind Klassen, deren Zweck einzig und allein darin besteht, Daten strukturiert zu speichern.

```
class Employee:
    def __init__(
        self,
        name: str,
        department: str = 'cs'
    ) -> None:
        self.name = name
        self.department = department

    def __str__(self) -> str:
        return 'Employee(name={}, department={})'.format(
            self.name,
            self.department
        )
```

Viel "Boilerplate"-Code — das lässt sich automatisieren!

DATENKLASSEN

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    department: str = 'cs'
```

`@dataclass` erzeugt automatisch die Attribute `name` und `department`, eine `__init__`-Methode, eine `__str__`-Methode und noch einige mehr.

In `Employee` sind `name` und `department` keine Klassenvariablen.

DATENKLASSEN

```
>>> print(Employee('Wolfgang Händler'))
Employee(name='Wolfgang Händler', department='cs')
```

DUCK TYPING

DUCK TYPING

> *"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."*
> *(James Whitcomb Riley)*

Die meisten Sprachen verwenden den Typ eines Objektes, um zu bestimmen, ob das Objekt in einem bestimmten Zusammenhang verwendet werden kann.

Python überprüft einfach zur Laufzeit, ob ein Objekt alle benötigten Methoden und Attribute hat. Diese Vorgehensweise nennt man Duck Typing.

KEIN DUCK TYPING: JAVA

Java (kein Duck Typing):

```
class Duck {
  void fly() { ... }
}

class Goose {
  void fly() { ... }
}
```

Zwei Klassen, beide haben eine Methode `fly`.

KEIN DUCK TYPING: JAVA

Die Methode `scare` erwartet einen Parameter `x` und ruft seine Methode `fly` auf.
Der Parameter muss den Typ `Duck` haben.

```
void scare(Duck x) {
  x.fly();
}
```

Versucht man, ein Objekt vom Typ `Goose` zu übergeben, schlägt das schon beim Übersetzen des Programs fehl — obwohl auch `Goose` die benötigte Methode `fly` hätte.

```
Goose goose = new Goose();
scare(goose); // Fehler: scare erwartet Duck, bekommt Goose
```

DUCK TYPING

Das gleiche Programm in Python:

```
class Duck:
  def fly(self): ...

class Goose:
  def fly(self): ...
```

```
def scare(x):
  x.fly()
```

```
goose = Goose()
scare(goose)  # ok
```

In Python läuft das Programm erst einmal bis zu dem Punkt, an dem die Methode `fly` des übergebenen Objektes aufgerufen wird: `x.fly()`
Erst jetzt wird kontrolliert, ob das Objekt eine passende Methode hat → ✓

DUCK TYPING: VOR- UND NACHTEILE

VORTEILE

NACHTEILE

DUCK TYPING: VOR- UND NACHTEILE

VORTEILE

- Flexibel: Man kann den für `Duck` geschriebenen Code einfach für `Goose` wiederverwenden.
- Einfache Typ-Strukturen.
  (Während man in Java ein zusätzliches *Interface* einführen würde, das die benötigten Methoden beschreibt. 🔬)

NACHTEILE

DUCK TYPING: VOR- UND NACHTEILE

VORTEILE

- Flexibel: Man kann den für `Duck` geschriebenen Code einfach für `Goose` wiederverwenden.
- Einfache Typ-Strukturen.
  (Während man in Java ein zusätzliches *Interface* einführen würde, das die benötigten Methoden beschreibt. 🔬)

NACHTEILE

```
def foo(x):
    if super_unwahrscheinliche_bedingung:
        x.diese_methode_gibt_es_gar_nicht()
```

Es ist im Allgemeinen nicht möglich, solche Fehler schon beim Übersetzen des Programmes zu finden. *Auch deshalb ist es in Python extrem wichtig, umfangreiche Tests für seine Software zu schreiben!*

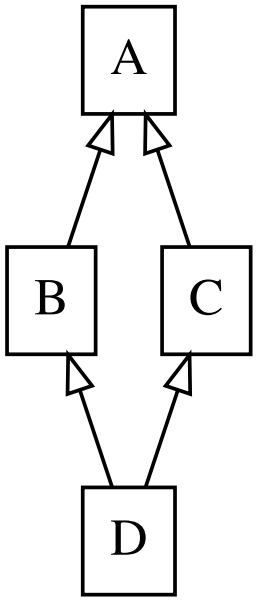
MEHRFACHVERERBUNG

MEHRFACHVERERBUNG

In Python kann eine Klasse von mehreren Elternklassen erben.

```
class DerivedClassName(Base1, Base2, Base3):
    pass
```

MEHRFACHVERERBUNG: SUPER




```python
class A:
    def __init__(self):
        pass

class B(A):
    def __init__(self):
        super().__init__()

class C(A):
    def __init__(self):
        super().__init__()

class D(B, C):
    def __init__(self):
        super().__init__()
```

- Jede der `__init__`-Methoden der abgeleiteten Klassen sollte einen `super`-Aufruf enthalten.
- Dann sorgt `super` dafür, dass jede der `__init__`-Methoden in der Hierarchie genau einmal und in der "richtigen" Reihenfolge ausgeführt wird.

MEHRFACHVERERBUNG: SUPER

⚠ In älteren Code-Beispielen sieht man häufig, dass auf die Methoden der Elternklassen ohne die Verwendung von `super` zugegriffen wird.
Das sollte man (in aller Regel) vermeiden!

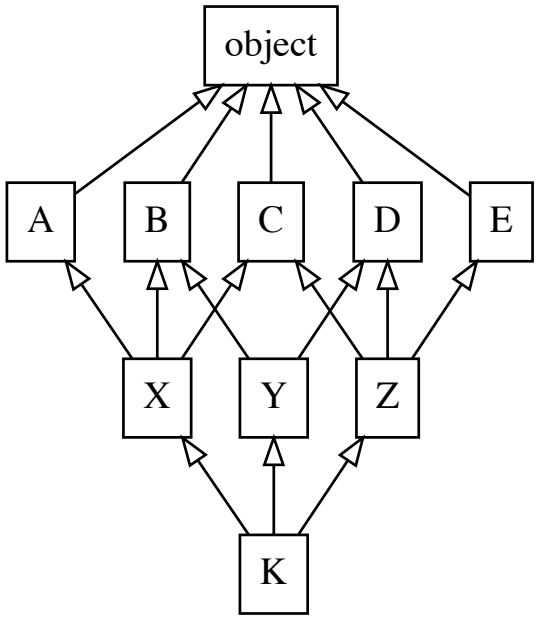
Falsch: 💀

```
class D(B, C):
    def __init__(self):
        B.__init__(self)
        C.__init__(self)
```

In unserem Beispiel würde die `__init__`-Methode von A damit doppelt aufgerufen!

MEHRFACHVERERBUNG 

Mehrfachvererbung kann unter Umständen aber sehr komplex werden.




```
class A: ...
class B: ...
class C: ...
class D: ...
class E: ...
class X(A, B, C): ...
class Y(B, D): ...
class Z(C, D, E): ...
class K(X, Y, Z): ...
```

In welcher Reihenfolge wird, ausgehend von K, in den Klassen nach einer Methode gesucht?

MEHRFACHVERERBUNG 🔬

Auf diese *Method Resolution Order* (MRO) kann man mit `K.mro()` zugreifen:

```
K, X, A, Y, B, Z, C, D, E, object
```

In Python wird diese Reihenfolge mit dem "C3 Superclass Linearization"-Algorithmus bestimmt.

Take-Home Message: Mehrfachvererbung ist komplex. Wenn man sie unbedingt verwenden will, sollte man sich genau überlegt haben, was man tut!

RUN-TIME TYPE INFORMATION

RTTI: ISINSTANCE

Mit der Funktion `isinstance` kann man überprüfen, ob ein Wert einen bestimmten Typ hat.

```
>>> s = 'abc'
>>> isinstance(s, str)
True
>>> isinstance(s, int)
False
```

RTTI: POLYMORPHIE

Auch ein Objekt einer abgeleiteten Klasse wird als Instanz des Elterntyps betrachtet: Jeder PKW ist auch ein Fahrzeug.

```
>>> f = PKW()
>>> isinstance(f, PKW)
True
>>> isinstance(f, LKW)
False
>>> isinstance(f, Fahrzeug)
True
```

Dieses OOP-Prinzip nennt man **Polymorphie** und es spielt eine zentrale Rolle in anderen objektorientierten Programmiersprachen.

RTTI: ISSUBCLASS

Ähnlich kann man mit `issubclass` überprüfen, ob eine Klasse von einer anderen Klasse abgeleitet ist.

```
>>> issubclass(PKW, Fahrzeug)
True
```

Eine Klasse wird als Unterklasse von sich selbst betrachtet:

```
>>> issubclass(int, int)
True
```

EXCEPTIONS

EXCEPTIONS

In gewissen Situationen können Programmfehler auftreten, die der Programmteil selbst nicht sinnvoll behandeln kann.

```
>>> 0 / 0
ZeroDivisionError: division by zero
```

```
>>> p = PKW()
>>> p.springen()
AttributeError: 'PKW' object has no attribute 'springen'
```

```
>>> open('bielefeld.txt')
FileNotFoundError: [Errno 2] No such file or directory: 'bielefe
```

EXCEPTIONS

In diesen Situationen wird eine *Exception* ausgelöst.

Die Exception wird modelliert durch eine von `Exception` abgeleitete Klasse.

Die Exception-Objekte enthalten eine Fehlermeldung und ggf. weitere Informationen.

EXCEPTIONS WERFEN

```
from typing import List

def maximum(a: List[int]) -> int:
    m = a[0]
    for x in a[1:]:
        if x > m:
            m = x
    return m

maximum([])
```

`maximum` funktioniert nur, wenn die übergebene Liste nicht leer ist,

Wie soll sich die Funktion verhalten, wenn `a` leer ist?

⇒ Die *aufrufende* Methode soll entscheiden, wie die Situation gehandhabt wird.

EXCEPTIONS WERFEN

```
def maximum(a: List[int]) -> int:
    if len(a) == 0:
        raise Exception('Die Liste ist leer!')

    m = a[0]
    for x in a[1:]:
        if x > m:
            m = x
    return m

maximum([])
```

EXCEPTIONS ABFANGEN

```
try:
    x = maximum(a)
except Exception as e:
    print('Warnung: Konnte kein Maximum bestimmen.')
    print('Verwende Standardwert.')
    x = 42
```

EIGENE EXCEPTION-KLASSEN

`Exception` ist die Basisklasse für *alle* Exceptions. Wir fangen also alle möglicherweise auftretenden Fehler ab, wollen aber eigentlich nur eine ganz bestimmte Fehlersituation behandeln.

Lösung: Man kann eine eigene Exception-Klasse anlegen, die nur in einer ganz bestimmten Situation verwendet wird.

```
class ListEmptyException(Exception):
    pass
```

Die Klasse `Exception` stellt schon alles bereit, was man normalerweise braucht, man muss nur von ihr erben.

EIGENE EXCEPTION-KLASSE

```
def maximum(a: List[int]) -> int:
    if len(a) == 0:
        raise ListEmptyException('Die Liste ist leer!')

    m = a[0]
    for x in a[1:]:
        if x > m:
            m = x
    return m
```

```
try:
    x = maximum(a)
except ListEmptyException as e:
    print('Leere Liste! Verwende Standardwert als Maximum.')
    x = 42
```