OBJEKTORIENTIERTE PROGRAMMIEREUNG

(OOP)

WAS IST OOP?
UND WELCHE VORTEILE BIETET ES?

- Programmierparadigma
- Strukturierung in Objekte mit fest definierten Eigenschaften
- Erstellung neuer Objekte aus „Vorlagen“ / Klassen

- Struktur ist semantisch sehr nahe an vielen Daten der „echten Welt“ und deren Kategorisierung
- Stark modular – leichte Wiederverwendung von Code
- Vereinfacht Troubleshooting bestimmter Probleme

KONVENTIONEN UND REGELN FÜR KLASSEN

- CamelCase
- Einzahl
- Gliederung durch Einrückung

```
class StudentRecord:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

KLASSE UND OBJEKT

KLASSE

- Bauplan/ Vorlage, mit der man Objekte erzeugt.
- beschreibt, welche
  - EIGENSCHAFTEN A(ATTRIBUTE) und
  - FÄHIGKEITEN (METHODEN)

  ein Objekt haben soll.

OBJEKT

- konkrete Instanz einer Klasse.
- Es ist ein tatsächlich existierendes „Ding“ mit eigenen Datenwerten.

ATTRIBUTE UND METHODEN

ATTRIBUTE

- Eigenschaften eines Objekts.
- Attribute werden meistens im Konstruktor initialisiert.
- (der schnelle Ferrari, die schwarze Katze,…)

METHODEN

- Methoden, sind Funktionen, die zur Klasse gehören
- (bellen, fahren, fliegen, …)

ATTRIBUTE UND METHODEN

ATTRIBUT

```
class Hund:
    def __init__(self, name, alter):
        self.name = name
        self.alter = alter
```

METHODE

```
class Hund:
    def __init__(self, name, alter):
        self.name = name
        self.alter = alter

    def bellen(self):
        print(f"{self.name} bellt!")
```

WARUM VERWENDET MAN „SELF“?

- **self** ist in Python eine REFERENZ auf das AKTUELLE OBJEKT

- Man greift damit innerhalb der Klasse auf die ATTRIBUTE und METHODEN DIESES OBJEKTS zu.

WIE VERWENDET MAN „SELF“

self.name = Attribut des Objekts

self greift auf das Objekt zu

```
class Hund:
    def __init__(self, name, alter):
        self.name = name
```

```
    def bellen(self):
        print(f"{self.name} bellt!")

    def info(self):
        print(f"{self.name} ist
        {self.alter} Jahre alt.")
```

ERSTELLEN EINER KLASSE

In Python wird sie mit dem Schlüsselwort **class** erstellt

- Der Konstruktor ist eine spezielle Methode, die beim Erstellen eines Objekts automatisch **immer** genau einmal aufgerufen wird.
- In Python besteht der Konstruktor immer zumindest aus __init__(self)
- Er dient dazu, Attribute zu initialisieren.

```
class Haustiere:
    #Konstruktor der Klasse Haustiere:
    def __init__(self,name,rasse):
        self.name = name
        self.rasse = rasse #Attribut Rasse
```

OBJEKT ERZEUGEN IN PYTHON

- Zuerst eine **Klasse definieren**
- Die Klasse **aufrufen wie eine Funktion**
- Dabei wird der **Konstruktor (__init__)** ausgeführt.

➔ `mein_hund = Hund("Bello", 3)`

→ `Hund("Bello", 3)` ruft den Konstruktor `__init__` auf

→ der Name wird mit `self.name = "Bello"` gesetzt

→ das Alter wird mit `self.alter = 3` gesetzt

➔ `mein_hund` ist jetzt ein Objekt der Klasse Hund

OBJEKT „ MEIN_HUND“ VERWENDEN

- `print(mein_hund.name)` `# Zugriff auf Attribut` → AUSGABE: Bello
- `print(mein_hund.alter)` → AUSGABE: 3
- `mein_hund.bellen()` `# Aufruf der Methode` → AUSGABE: Bello bellt!

BEISPIEL (OOP)

```
class Hund:
      def __init__(self, name, alter):
            self.name = name
            self.alter = alter

def bellen(self):
      print(f„Mein Hund heißt {self.name}, ist {self.alter}
      Jahre alt und bellt!")

mein_hund = Hund("Bello", 3)                # Objekt erzeugen


mein_hund.bellen()                          # Methode aufrufen
```

AUSGABE:

```
Mein Hund heißt Bello, ist 3 Jahre alt
und bellt!"
```

VERERBUNG UND SUPERKLASSEN

OBJEKTORIENTIERTE PROGRAMMIERUNG

SUPERKLASSEN & UNTERKLASSEN

- **SUPERKLASSE** = Allgemeiner Bauplan
  - **allgemeine Klasse**, von der andere Klassen erben können
  - enthält Eigenschaften(Attribute) und Methoden, die **alle Unterklassen gemeinsam** haben
  - Beispiel: **Alle Haustiere** haben einen **Namen** und ein **Alter**

- **UNTERKLASSE** = Spezialisierung des Bauplans
  - bekommt alle Attribute und Methoden der Superklasse automatisch
  - kann eigene Methoden hinzufügen oder geerbte Methoden überschreiben
  - Jede Unterklasse kann ihr eigenes Verhalten definieren.

ERZEUGEN DER SUPERKLASSE

SCHLÜSSELWORT SUPER()

- **super()** ruft Methoden der SUPERKLASSE auf

```
class Haustier:
    def __init__(self, name, alter):
        self.name = name
        self.alter = alter

    def info(self):
        print(f"{self.name} ist
        {self.alter} Jahre alt.")
```

ERZEUGEN DER UNTERKLASSE(N)

```
class Hund(Haustier):
    def geraeusch(self):
        print(f"{self.name} bellt: Wuff!")

class Katze(Haustier):
    def geraeusch(self):
        print(f"{self.name} miaut: Miau!")

    def klettern(self):
        print(f"{self.name} klettert auf einen
        Baum.")
```

ERZEUGEN DER UNTERKLASSE(N)

```
class Hund(Haustier):
    def __init__(self, name, alter, rasse):
        super().__init__(name, alter)                # ruft Konstruktor der Superklasse auf
        self.rasse = rasse                           # eigenes Attribut der Unterklasse

    def spielen(self):
        print(f"{self.name}, ein {self.rasse}, spielt gerne mit einem Ball.")


class Katze(Haustier):
    def __init__(self, name, alter, fellfarbe):
        super().__init__(name, alter)
        self.fellfarbe = fellfarbe

    def klettern(self):
        print(f"{self.name} mit {self.fellfarbe}em Fell klettert flink auf einen Baum.")
```

OBJEKTE ERZEUGEN (SUPERKLASSE)

OBJEKTE ERZEUGEN

```
bello = Hund("Bello", 3, "Dackel")
luna = Katze("Luna", 2, "schwarz")
```

GEMEINSAME METHODE AUS DER SUPERKLASSE

```
bello.info()   → Bello ist 3 Jahre alt.
luna.info()    → Luna ist 2 Jahre alt.
```

UNTERSCHIEDLICHES VERHALTEN (METHODE)

```
bello.spielen()

➔  Bello, ein Dackel, spielt gerne mit dem Ball.

luna.klettern()

➔  Luna mit schwarzem Fell klettert flink auf einen Baum.
```

BEISPIEL FÜR SUPERKLASSE & UNTERKLASSE

```
class Hund(Haustier):
    def __init__(self, name, alter, rasse):

super().__init__(name, alter)                    # Konstruktor der Superklasse aufrufen
        self.rasse = rasse                       # eigenes Attribut der Unterklasse

    def info(self):
        print(f"{self.name} ist ein {self.rasse} und {self.alter}")
```

CODEBEISPIEL FÜR SUPERKLASSE & UNTERKLASSE

```
class Haustier:                                          # Superklasse
    def __init__(self, name, alter):
        self.name = name
        self.alter = alter

    def info(self):
        print(f"{self.name} ist {self.alter} Jahre alt.")


class Hund(Haustier):                                    # Unterklasse Hund
    def __init__(self, name, alter, rasse):
        # ruft den Konstruktor der Superklasse auf
        super().__init__(name, alter)
        self.rasse = rasse                               # eigenes Attribut der Unterklasse

    def spielen(self):
        print(f"{self.name}, ein {self.rasse}, spielt gerne mit dem Ball.")
```

```
# HUND-OBJEKT „BELLO“ ERZEUGEN
bello = Hund("Bello", 3, "Golden Retriever")

# AUFRUF DER GEERBTEN METHODE AUS DER SUPERKLASSE
bello.info()        → Bello ist 3 Jahre alt.

# AUFRUF DER EIGENEN METODE AUS DER UNTERKLASSE
bello.spielen()     → Bello, ein Golden Retriever, spielt
                      gerne mit dem Ball.
```

VORTEILE VON VERERBUNG UND SUPERGRUPPEN

- **klare Strukturierung der Klassen** → übersichtlicher

- **Wiederverwendbarkeit** durch Vererbung
  - Der Code der Superklasse kann wiederverwendet werden und gilt auch für alle Unterklassen
  - Die Initialisierung (oder andere Methoden) der Superklasse kann weiterverwendet werden

- **leichte Erweiterbarkeit**, wenn neue Funktionen oder Objekte hinzukommen
  - Es können neue Methoden und Attribute in der Superklasse ergänzt werden, die für alle Unterklassen gelten
  - Es können Unterklassen ergänzt werden, die die Eigenschaften und Methoden der Superklasse erben

ASSOZIATION UND VERERBUNG

ASSOZIATION

- „hat ein“ oder „kennt ein“
- Klassen interagieren miteinander bzw. haben eine Beziehung zueinander
- Eine Klasse benutzt eine andere Klasse, indem sie eine Instanz davon enthält
- Keine geerbten Eigenschaften

VERERBUNG

- „ist ein“
- Eine Unterklasse ist eine Spezialisierung der Oberklasse
- Methoden/Attribute der Oberklasse werden geerbt

BEISPIEL FÜR ASSIOZIATION

```
class Lehrer:
    def __init__(self, name):
        self.name = name

    def unterrichten(self, schueler):
        print(f„Der Lehrer {self.name} unterrichtet {schueler.name}.")

class Schueler:
    def __init__(self, name, lehrer):
        self.name = name
 # Assoziation: Der Schüler "kennt" einen Lehrer
        self.lehrer = lehrer

    def vorstellung(self):
        print(f"Ich bin {self.name} und mein Lehrer ist {self.lehrer.name}.")

lehrer1 = Lehrer("Herr Korber")
schueler1 = Schueler("Max", lehrer1)

schueler1.vorstellung()
schueler1.lehrer.unterrichten(schueler1)
```

`lehrer1 = Lehrer("Herr Korber")`

→ erzeugt ein Objekt vom Typ Lehrer mit dem Namen „Herr Korber“

`schueler1 = Schueler("Max", lehrer1)`

→ erzeugt ein Objekt vom Typ `Schueler` mit dem Namen `"Max"` und einer Referenz (`lehrer`) auf das Lehrer-Objekt `lehrer1`.

`schueler1.vorstellung()`

→ **„Ich bin Max und mein Lehrer ist Herr Korber.“**

`schueler1.lehrer.unterrichten(schueler1)`

→ `schueler1.lehrer` greift auf das Attribut `lehrer` von `schueler1` zu

→ Bezieht sich auf das Objekt `lehrer1` (vom Typ `Lehrer`)

→ `unterrichten()` → ruft die Methode `unterrichten()` der Klasse `Lehrer` auf

→ **Der Lehrer Herr Korber unterrichtet Max.**

KAPSELUNG

OBJEKTORIENTIERTE PROGRAMMIERUNG

VORTEILE VON KAPSELUNG

- DATEN (ATTRIBUTE) und die dazugehörigen OPERATIONEN (METHODEN) werden in einer Klasse zusammengefasst.

- Der DIREKTE ZUGRIFF auf interne Daten wird EINGESCHRÄNKT
  Außenstehende können nur über definierte Schnittstellen/ Methoden mit dem Objekt interagieren.

- **ZIEL:**
  - → Schutz der Daten vor unkontrollierten Zugriffen und Manipulation
  - → Vermeidung von Fehlern – Außenstehende können interne Variablen nicht beliebig ändern
  - → Erhöht Sicherheit, Robustheit und Wartbarkeit

ERZEUGEN DER KAPSELUNG

```
class Konto:
    def __init__(self, inhaber, saldo):
        self.inhaber = inhaber
        self.__saldo = saldo                    # private Variable

    def einzahlen(self, betrag):
        self.__saldo += betrag                  # verändern schwierig
```

VERGLEICH: MIT UND OHNE KAPSELUNG

MIT KAPSELUNG

```
class Konto:
    def __init__(self, inhaber, saldo):
        self.inhaber = inhaber
        self.__saldo = saldo                          # private Variable

    def einzahlen(self, betrag):
        self.__saldo += betrag

    def abheben(self, betrag):
        if betrag <= self.__saldo:
            self.__saldo -= betrag
        else:
           print("Nicht genug Guthaben!")

    def kontostand(self):
        return self.__saldo
```

OHNE KAPSELUNG

```
class Konto:
    def __init__(self, inhaber, saldo):
        self.inhaber = inhaber
        self.saldo = saldo       # direkt von außen zugreifbar

→ PROBLEM: Jeder kann einfach das Guthaben manipulieren

konto = Konto("Anna", 1000)
konto.saldo = -5000              # Unsinniger Wert!
```