Objektorientierte Programmierung 3

OOP

Erben auch von eingebauten Klassen

- ererben Methoden
- ererben Attribute
- können erweitern

OOP

Beispiel sortierte Liste (Buch 19.10)

```
1  class SortierteListe(list):
2
3      def __init__(self, *args, **kwargs):
4          super().__init__(*args, **kwargs)
5          self.sort()
6
7      def __setitem__(self, key, value):
8          super().__setitem__(key, value)
9          self.sort()
10
11     def append(self, value):
12         super().append(value)
13         self.sort()
```

wo Änderung: zusätzlich sort()

OOP

sortierte Liste ermöglicht Suche wie Binärbaum:

- wähle das mittlere Element der Liste
- Vergleiche.
  - ist Suchelement kleiner: Suche in vorderer Hälfte
  - ist Suchelement größer: Suche in hinterer Hälfte
  - Suchelement gleich: gefunden

OOP

Fortgeschrittene Aufgabe

Implementieren Sie für SortierteListe eine Methode

bin_suche(), die den vorstehenden Algorithmus umsetzt

- rekursiv
- nicht slices übergeben sondern Indizes
- optional: Zeit messen und mit "in" vergleichen

OOP

OOP bisher:

- Definition von Klassen
- imperativer Aufruf von Methoden

- wie sieht OO-Ablauf aus?

OOP

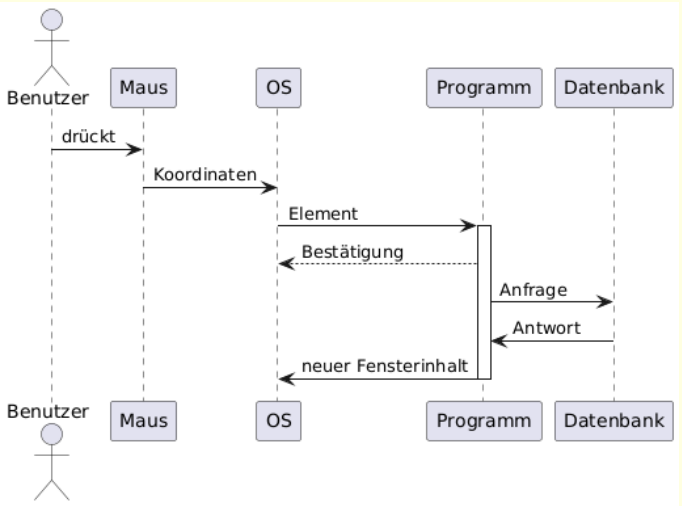
Austausch von Nachrichten über

- Methodenaufrufe
- Rückgabewerte

OOP

Austausch von Nachrichten

```
1  class maus:
2
3
4
5      def linker_knopf(self):
6          # irgendwas
7          os.mausklick( 22, 33 )
8
9  class OS:
10
11
12
13     def mausklick( self, x, y ):
14         pass
```



OOP

Austausch von Nachrichten über

```
1  class maus:
2      def __init__(self, os):
3          self.mein_os = os
4
5      def linker_knopf(self):
6          # irgendwas
7          self.mein_os.mausklick( 22, 33 )
8
9  class OS:
10     def neue_maus( self ):
11         mausliste.append( maus( self ) )
12
13     def mausklick( self, x, y ):
14         pass
```

Maus und OS
müssen sich kennen

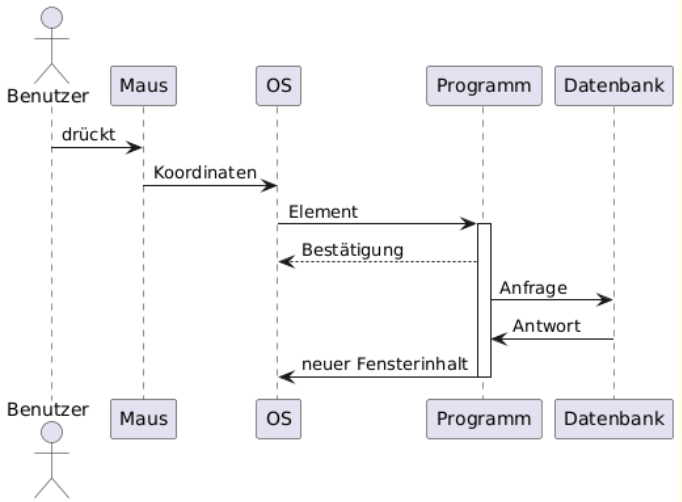
OOP

Austausch von Nachrichten

```python
class programm:
    def neue_db(self, db):
        self.meine_db = db

    def verarbeitung(self):
        # ...
        daten = self.meine_db.query( sql )
        # ...

class datenbank:
    def query( self ):
        # query verarbeiten
        return antwort
```



OOP

Austausch von Nachrichten

```
1  class programm:
2      def neue_db(self, db):
3          self.meine_db = db
4
5      def verarbeitung(self):
6          # ...
7          daten = self.meine_db.query( sql )
8          # ...
9
10 class datenbank:
11     def query( self ):
12         # query verarbeiten
13         return antwort
```

- Datenbank kennt Programm nicht
- verarbeitung() muss auf Antwort warten

OOP

Austausch von Nachrichten

```
1  class programm:
2      def neue_db(self, db):
3          self.meine_db = db
4
5      def verarbeitung(self):
6          # ...
7          self.meine_db.query( sql, self )
8
9      def query_ergebnis(self, antwort):
10         # Verarbeitung der Antwort
11
12 class datenbank:
13     def query( self, sql, anfrager):
14         # query verarbeiten
15         anfrager.query_ergebnis( antwort )
```

ohne Warten
auf Antwort

OOP

Austausch von

Nachrichten

über

Methodenaufrufe

```
1  class Hund(Tier):
2      ...
3      def provozieren( self, wer ):
4          return wer.beissen()
5
6  class Katze(Tier):
7      ...
8      def sieht_hund( self, hund ):
9          return hund.provozieren( self )
10
11     def beissen(self):
12         return "Aua!"

hu=Hund("Bello", 3645)
ka=Katze("Mia","braun")

print(ka.sieht_hund( hu ))
```

OOP

Fügen Sie Tiere.py eine Klasse *Mensch* hinzu.

- der Konstruktor nimmt als Parameter eine Liste von 1-5 Tieren, die dem Menschen gehören
- die Tiere werden in einer Menge in der Mensch-Instanz gespeichert
- Mensch hat eine Methode *anbetteln();* wird diese von einem Tier aufgerufen so
  - ruft sie dessen *fuettern()*-Methode auf, wenn das Tier ein Hund ist, der dem Menschen gehört