10 Mer om klasser og objekter

Læringsmål dette kapittel

- Forklare er-en og har-en relasjoner mellom klasser og objekter
- Implementere komposisjon og aggregat i Python
- Bruke objektreferanser for å modellere relasjoner
- Forklare og implementere arv med super() og konstruktørkall
- Overstyre metoder i subklasser og forklare hensikten
- Demonstrere polymorfi med felles metoder i ulike subklasser
- Beskrive formålet med UML og klassediagrammer
- Tolke og lage UML-diagrammer med klasser, attributter og metoder
- Bruke UML-syntaks for synlighet og struktur
- Forklare abstrakte klasser og hvorfor de brukes
- Bruke abc-modulen til å definere abstrakte metoder
- Implementere abstrakte metoder i subklasser
- Forklare og implementere Iterator pattern i Python
- Bruke __iter__ og __next__ i egne klasser
- Forklare duck typing og hvordan polymorfi kan oppnås uten arv
- Bruke isinstance() for typekontroll og spesiallogikk
- Utvide Account-klassen til et klassehierarki med subklasser
- Implementere en Bank-klasse som håndterer ulike kontotyper
- Skrive klientkode som bruker arv, polymorfi og typekontroll

10.1 Innledning

Vi bruker klasser for å modellere / abstrahere «ting» / entiteter fra «den virkelige verden». Når vi opererer med mer enn en klasse vil det gjerne være slik at det oppstår *relasjoner* mellom klassene og objektene. Det er spesielt to typer relasjoner som er viktige å kjenne til mellom klasser og objekter;

er-en relasjon (også kalt ***arv***), en relasjon mellom klasser

har-en relasjon, som kan være to typer: komposisjon og aggregat, en relasjon mellom objekter

© Frode Næsje

10.2 Har-en (hel-del) relasjon: komposisjon og aggregat

Denne typen relasjon oppstår når et objekt består av ulike deler, eller at et objekt inneholder en samling av andre objekter.

En bil består av en motor, hjul, ratt, har en farge, etc. Det er sannsynlig at vi ønsker å lage *motor* som en egen klasse, sånn at vi kan utstyre bilen med ulike motorer:

```
class Engine:
    def __init__(self, horsepower):
        self._horsepower = horsepower
class Car:
    def __init__(self, horsepower):
        self._engine = Engine(horsepower)

# Opprette en bil med en motor
car = Car(150)

# Hvis bilen slettes, slettes også motoren
del car # Motoren eksisterer ikke lenger
```

Koden over viser en ***har-en*** (også kalt hel-del) relasjon av typen ***komposisjon***, som betyr:

- Delene er helt avhengige av helheten.
- Når helheten slettes, slettes også delene.
- En klasse "Car" kan ha objekter som "Engine" og "Wheel". Hvis bilen slettes, gir det ikke mening at motoren eller hjulene eksisterer alene.

Aggregat er en annen, ikke så streng ***har-en*** relasjon, hvor

- Delene kan eksistere uten helheten
- Helheten og delene har en **løs kobling**

Eksempel: En klasse `Team` kan ha en liste med `Person`-objekter som medlemmer. Hvis teamet slettes, kan personene fortsatt eksistere:

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

class Team:
```

© Frode Næsje

```
    def __init__(self):
        self._members = []
    def add_member(self, person):
        self._members.append(person)

# Opprette personer
person1 = Person("Alice")
person2 = Person("Bob")

# Opprette et team og legge til medlemmer
team = Team()
team.add_member(person1)
team.add_member(person2)

# Selv om teamet slettes, eksisterer personene
fortsatt
del team
print(person1.name)  # Output: Alice
```

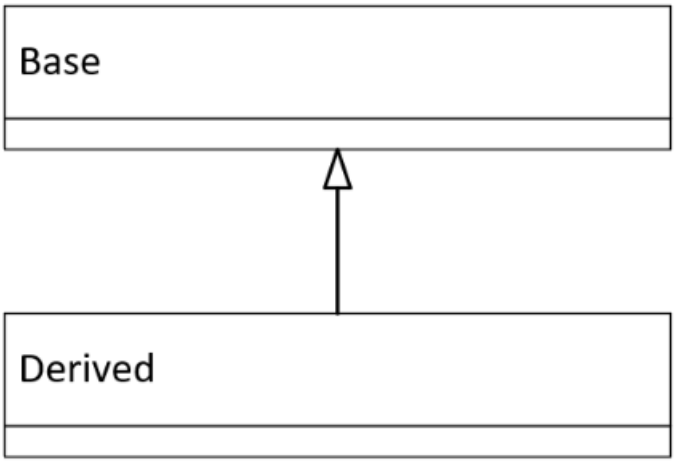
Merk: uansett type ***har-en*** relasjon (komposisjon eller aggregat), så ser vi at selve relasjonen er implementert ved at det ene objektet har en referanse til (en eller flere) av den andre objekt typen.

10.3 Er-en relasjon: Arv

Arv er et grunnleggende konsept i objektorientert programmering som lar en klasse (kalt *subklasse* eller *barnklasse*) *arve* egenskaper og metoder fra en annen klasse (kalt *superklasse* eller *foreldreklasse*). Dette gjør det mulig å gjenbruke kode og skape hierarkier av klasser.

Arv illustreres slik i et UML diagram (Derived arver Base)

© Frode Næsje



Arv-mekanismen lar oss bruke egenskaper (attributter) og metoder fra en annen klasse, samtidig som vi kan *legge til* eller *overstyre* funksjonalitet.

Når en klasse arver fra en annen klasse:

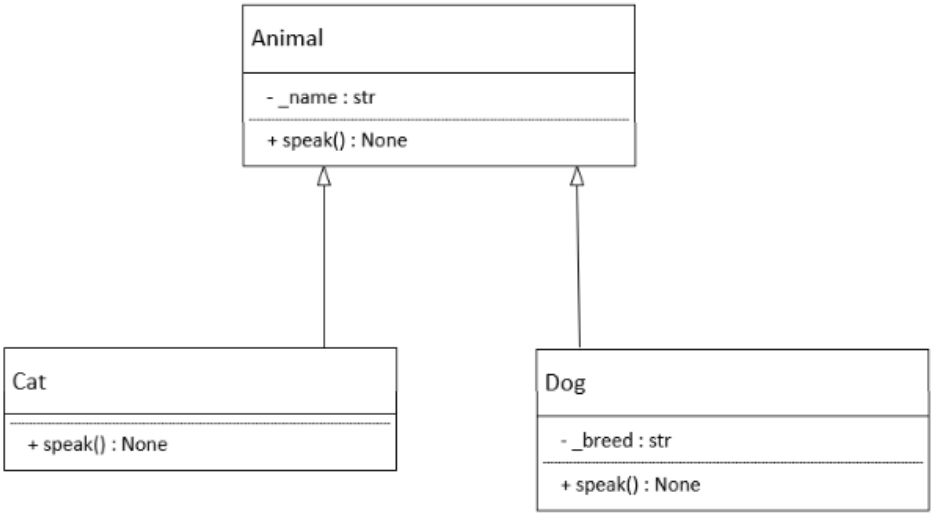
- Den nye klassen (subklassen) får tilgang til alle attributter og metoder fra superklassen
- Subklassen kan *legge til* nye attributter og metoder.
- Subklassen kan *overstyre* (override) eksisterende metoder fra superklassen.

Følgende kodeeksempel viser en klasse `Animal` med barneklasser `Dog` og `Cat`. Klassen `Dog` har en egen attributt `_breed`, og både `Dog` og `Cat` har sin egen variant av metoden `speak`. Andre detaljer i koden viser hvordan vi sier at en klasse skal arve en annen klasse, og hvordan vi kaller foreldrekonstruktør fra barneklasse.

UML diagram:

- + betyr at attributten eller metoden er public, altså en del av grensesnittet (tilgjengelig for klientkode)
- betyr at attributten eller metoden er privat, skal ikke aksesseres av klientkode

© Frode Næsje



Se forklaring etter koden:

```
1: class Animal:  # Foreldreklasse, arver fra object
2:     def __init__(self, name):
3:         self._name = name
4:
5:     def speak(self):
6:         return "Some generic animal sound"
7:
8: class Dog(Animal):  # Subklasse av Animal
9:     def __init__(self, name, breed):
10:         super().__init__(name)
11:         self._breed = breed
12:
13:     def speak(self): # Overstyrer speak fra Animal
14:         return "Woof!"
15:
16: class Cat(Animal):  # Subklasse av Animal
17:     def speak(self): # Overstyrer speak fra Animal
18:         return "Meow!"
19:
20: # Polymorfi i praksis:
21: animals = [Dog("Fido", "Labrador"), Cat("Misty"),
Animal("Generic")]
22:
23: for animal in animals:
24:     print(f"{animal._name} sier: {animal.speak()}")
```

© Frode Næsje

`Utskrift:`
Fido sier: Woof!
Misty sier: Meow!
Generic sier: Some generic animal sound

Forklaring til kodelinjene:

Linje 1 - 7:
`Animal` er foreldreklassen i dette arvehierarkiet. Den har en attributt `self._name` , og en metode `speak(self)`.*`Animal` arver implisitt (automatisk) fra klassen `Object`, som er den ultimate foreldreklassen for alle Python objekter.*

Linje 8 – 14

Dette er en subklasse `Dog` som *arver* fra `Animal`. Merk at vi arver `Animal` ved å bruke parantesen i klassenavnet:

`class Dog(Animal):`

`Dog` utvider funksjonaliteten ved å legge til en ny attributt `_breed` (rase) i konstruktøren. Det totale `Dog` objektet består av attributtene `_name` og `_breed`.

`__init__`-metoden tar både `_name` og `_breed` som argumenter.

`super().__init__(name)` er et kall til konstruktøren i `Animal`, dette er nøvendig for å sette `self._name` i `Animal`.

`self._breed = breed` lagrer rasen som en privat attributt i `Dog` klassen.

`speak`-metoden overstyrer den generiske versjonen fra `Animal`, og returnerer "Woof!".

Linje 16–18:

Subklassen `Cat` arver også fra `Animal`, men har ikke en egen `__init__`-metode – ***den arver og bruker den fra `Animal`***:

Vi trenger å lage en egen `__init__()` for subklassen kun hvis:

- vi må initialisere subklassens egne, nye attributter

- endre default verdier

- gjøre validering

© Frode Næsje

Merk:

Hvis vi lager en egen `__init__()`, så, *må* vi utføre `super().__init__()`

`speak`-metoden overstyrer den generiske versjonen og returnerer "Meow!".

Linje 20–24:

Her demonstreres *polymorfi*, altså at objekter av ulike klasser kan behandles likt, men oppføre seg forskjellig.

Linje 21: En liste `animals` opprettes med én `Dog`, én `Cat`, og én `Animal`.

Linje 23–24: For-løkken itererer over listen og kaller `speak()` på hvert objekt. *Selv om alle objektene behandles som `Animal`, vil den riktige `speak`-metoden bli kalt basert på objektets faktiske klasse.*

10.4 UML diagrammer

Når vi utvikler programvare med objektorientert design, er det viktig å kunne visualisere strukturen og samspillet mellom klasser. Unified Modeling Language (UML) er et standardisert visuelt språk som brukes til å modellere programvarearkitektur, spesielt i objektorienterte systemer. UML hjelper utviklere, designere og dokumentasjonsforfattere med å forstå og kommunisere systemets struktur og oppførsel.

Klassediagrammer er en av de mest brukte UML diagrammene. Klassediagrammer viser klasser, attributter, metoder og relasjoner mellom klasser, og gir et oversiktlig bilde av systemets statiske struktur.

UTKAST

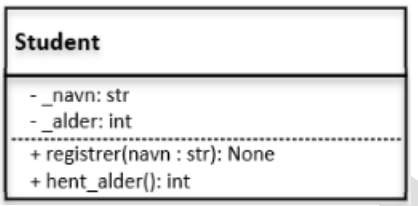
© Frode Næsje

Grunnleggende syntaks i UML-klassediagrammer

En klasse representeres som et rektangel delt inn i tre seksjoner:

1. **Klassenavn** – øverst, ofte med fet skrift
2. **Attributter** – i midten, med datatype og synlighet
3. **Metoder** – nederst, med parameterliste og returtype

Eksempel:



Her betyr:

- at attributten/metoden er **privat**
+ at den er **offentlig**
`None`, `int` etc er **datatyper** og **returtyper**

Relasjoner mellom klasser

UML støtter flere typer relasjoner:

- **Assosiasjon**: En klasse refererer til en annen
- **Aggregering**: En "har-del-av"-relasjon, men delene kan eksistere uavhengig
- **Komposisjon**: En sterkere "har-del-av"-relasjon, der delene ikke kan eksistere uten helheten
- **Arv (generalisering)**: En klasse arver egenskaper fra en annen
- **Avhengighet**: En klasse bruker en annen, men er ikke direkte koblet

<< Lag eksempler >>

UTKAST

© Frode Næsje

10.5 super()

`super()` metoden brukes når vi har behov for å kalle en metode i foreldreklassen *som heter det samme som en metode i barneklassen*.

Et typisk tilfelle er når vi ønsker å bruke foreldreklassens `__str__()` metode, i stedet for å skrive redundant kode. Her et eksempel som viser `super()` brukt både i *constructorkall* (`super().__init__(name, age)`) og fra barneklassens `__str__()`:

```
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    def __str__(self):
        return f"Navn: {self._name}, Alder: {self._age}"

class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self._student_id = student_id

    def __str__(self):
        # Bruker __str__ fra Person og legger til
student_id
        return super().__str__() + f", Student-ID:
{self._student_id}"

# Eksempelbruk
p = Person("Kari", 45)
s = Student("Ola", 22, "s12345")

print(p)  # Utskrift: Navn: Kari, Alder: 45, Student-ID:
s12345
```

10.6 Multippel arv

<< kan designmessig ofte erstattes av mix-in klasser og abstrakte klasser / interfaces, men må nevnes>>

© Frode Næsje

10.7 Interfaces / abstrakte klasser

"Interfaces define *what* should be done, not *how*.", eller "Interfaces are contracts, not implementations."

Dette uttrykker ideen om at et interface (eller en abstrakt baseklasse i Python) *beskriver hvilke metoder som må finnes, men ikke hvordan de skal implementeres.* Det er opp til hver subklasse å gi sin egen konkrete implementasjon.

Python har ikke interfaces, slik som en del andre språk, men ved hjelp av mekanismen *abstrakte klasser* kan vi oppnå det samme.

En abstrakt klasse brukes til å definere et felles grensesnitt for en gruppe relaterte klasser. En abstrakt klasse kan ikke instansieres direkte *og er ment å bli arvet av andre klasser*. Abstrakte klasser kan inneholde både abstrakte metoder (metoder uten implementasjon) og konkrete metoder (metoder med implementasjon).

Dersom du arver en abstrakt klasse aksepterer du at du må implementere de metodene som er abstrakte, og du må implementere metodene slik at de gjør det som er formålet.

Hvorfor kan noe slikt være nyttig?

La oss se litt tilbake på Animal klassen:

Egentlig er det ingen mening i at klassen `Animal` skal kunne *si noe* (metoden `speak()`). `Animal` er *generisk*, altså et ikke-eksisterende dyr, som kun er der fordi vi ønsker at et `Dog` og et `Cat` objekt skal oppføre seg forskjellig når vi kaller metoden `speak()` på slike objekter. For å få det til, må begge klassene arve `Animal`, samt implementere `speak()`.

I et slik tilfelle bør vi heller lage en abstrakt klasse, og bruke modulen `abc`.

10.7.1 Hva er abc modulen

abc-modulen i Python står for Abstract Base Classes, og brukes til å definere abstrakte klasser og metoder – altså klasser som fungerer som *maler* for andre klasser.

Kort forklart:

© Frode Næsje

- ABC er en baseklasse du arver fra for å lage en abstrakt klasse.
- @abstractmethod brukes for å markere metoder som ***må*** implementeres i subklasser.
- Du kan ikke instansiere en klasse som har abstrakte metoder.

Hvorfor bruke det?

- For å tvinge en struktur i subklasser.
- For å sikre at visse metoder alltid finnes i alle underklasser.
- For å lage interfaces i Python (selv om språket ikke har egne interface-klasser).

La oss se hvordan `Animal`-klassehierarkiet blir implementert med hjelp av **abc** modulen:

```
01: from abc import ABC, abstractmethod
02:
03: class Animal(ABC):
04:     def __init__(self, name):
05:         self.name = name
06:
07:     @abstractmethod
08:     def speak(self):
09:         pass
10:
11:     @abstractmethod
12:     def movement(self):
13:         pass
14:
15: class Dog(Animal):  # Subklasse av Animal
16:     def __init__(self, name, breed):
17:         super().__init__(name)
18:         self.breed = breed
19:
20:     def speak(self):  # implementerer speak
21:         return "Woof!"
22:
23:     def movement(self): # implementerer movement
```

© Frode Næsje

```
24:         return "Run"
25:
26: class Cat(Animal):  # Subklasse av Animal
27:     def speak(self):  # implementerer speak
28:         return "Meow!"
29:
30:     def movement(self):  # implementerer movement
31:         return "Sneak"
32:
33: animals = [Dog("Fido", "Labrador"), Cat("Misty")]
34:
35: for animal in animals:
36:     print(f"{animal.name} sier: {animal.speak()} b
evegelse: {animal.movement()}")
37:
38: another_animal = Animal("ImpossibleAnimal")
39: another_animal.speak()
```

Utskrift:

```
Fido sier: Woof! bevegelse: Run
Misty sier: Meow! bevegelse: Sneak
Traceback (most recent call last):
  File "c:\Users\fna003\OneDrive - UiT Office
365\Faglig\Python\Book\Code\tb_sc\ch_10_class_inheritance_p
olymorphism\sc_10_02_abstract_Animal.py", line 38, in
<module>
    another_animal = Animal("ImpossibleAnimal")
TypeError: Can't instantiate abstract class Animal with
abstract methods movement, speak
```

Forklaring til kode:
Linje 1:

Her importerer vi abc modulen. Fordi denne importeres kan vi arve ABC klassen, samt at vi kan anvende dekoratøren @abstractmethod. Vi får heller ikke lov til å lage objekter av Animal, kun barneklasser.

Linje 7 – 13:

Her annoterer vi metodene speak() og movement() med dekoratøren @abstractmethod; - dette sikrer at barneklasser av Animal må implementere disse metodene.

© Frode Næsje

Linje 15 – 31:

Barneklassene implementerer speak() og movement(), kontrakten er oppfylt!

Linje 33 – 36:

Vi oppretter et Dog og Cat objekt og kaller de polymorfe metodene.

Linje 38 og 39

Programmet krasjer på linje 38, det at vi arver ABC i Animal klassen gjør at vi ikke får instansiert et objekt av denne typen. Linje 39 kjøres ikke, fordi programmet har stoppet.

10.7.2 Den abstrakte klassen `Iterator`

Design patterns er et begrep innenfor «computer science» som går ut på at mange problemer egentlig har en felles løsning. Et velkjent design pattern er «Iterator pattern». Dette design pattern handler om hvordan en kan gjennomløpe en collection / samling data på en uniform måte, uansett hvordan collection'en ser ut. Dette design pattern er årsaken til at løkker av typen under fungerer:

```
for elem in [20, 30, 3, 5]:
     <din kode>
```

Bak kulissene er et såkalt *Iterator pattern* implementert for `list` klassen, som betyr at `list` klassen har implementert metodene `__iter__()` og `__next__()`.

`__iter__()` returnerer en *iterator*, mens `__next__()` returnerer *neste element* i collection'en

Python har i modulen Collections.abc implementert en abstrakt klasse som implementerer dette design pattern. Det vi må gjøre er å implementere `__iter__()` og `__next__()` for vår egen collection klasse.

Eksempelet under implementerer `Iterator` interfacet. Ved å importere den abstrakte klassen `Iterator` fra `collections.abc`, så vil python interpreteren sjekke at begge metodene er implementert i vår klasse `MyIterator` når det instansieres et objekt av den typen.

© Frode Næsje

```
# abstract_base_class_iterator.py
from collections.abc import Iterator

class MyIterator(Iterator):
    def __init__(self):
        self.data = [1, 2, 3]
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.data):
            item = self.data[self.index]
            self.index += 1
            return item
        else:
            raise StopIteration

my_iterator = MyIterator()
for item in my_iterator:
    print(item)
```

Utskrift:

```
1
2
3
```

Den abstrakte klassen dikterer altså at subklassen ***må*** implementere disse to metodene. I tilllegg må altså programmereren lese seg opp på hva disse metodene skal gjøre / levere.

Dette er som regel hensikten med en abstrakt klasse, å implementere en avtalt protokoll.

10.7.3 Python; - arv, polymorfi og duck typing

Det er faktisk ikke et absolutt krav i Python at polymorfi *må* bruke et klassehierarki eller en felles base class. Python støtter såkalt "duck typing": Hvis to (eller flere) objekter har metoder med samme navn og signatur, kan de brukes om hverandre – uavhengig av arv.

© Frode Næsje

- I klassisk OOP (som Java/C++), kreves ofte et felles klassehierarki for polymorfi.
- I Python er det vanligst og mest ryddig å bruke en felles base class når du vil ha strukturert polymorfi, spesielt i større systemer.
- Uten felles base class kan vi likevel bruke polymorfi så lenge objektene har de nødvendige metodene.

Eksempel på polymorfi uten arv:

```
class Dog:
    def speak(self): return "Woof!"
class Cat:
    def speak(self): return "Meow!"

for animal in [Dog(), Cat()]:
    print(animal.speak())  # Fungerer selv uten felles
base class
```

Utskrift:

```
Woof
Meow
```

Best practice for polymorf oppførsel er:

- Bruk felles base class (gjerne abstrakt base class med `abc.ABC` og `@abstractmethod`) når du ønsker struktur, feilsjekk og tydelig API for polymorfe objekter
- Bruk duck typing (ingen arv, bare samme metode-navn) for enkle eller små systemer der fleksibilitet er viktigere enn streng struktur
- Dokumenter hvilke metoder som forventes for polymorfe objekter, spesielt hvis du ikke bruker arv
- Foretrekk arv og abstrakte metoder når du vil sikre at alle subklasser implementerer nødvendige metoder

© Frode Næsje

10.7.4isinstance metoden

`isinstance()` er en innebygd funksjon i Python som brukes for å sjekke om et objekt er en instans av en bestemt klasse (eller en subklasse av denne). Dette er nyttig når du har en liste med ulike objekter, og du vil utføre spesielle operasjoner kun på objekter av en bestemt type.

Eksempel:

```
class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):
    def fetch(self):
        return "Fetching stick!"

class Cat(Animal):
    pass

animals = [Dog(), Cat(), Animal()]

for animal in animals:
    print(animal.speak())
    if isinstance(animal, Dog):
        print(animal.fetch())  # Bare Dog har fetch-
metoden
```

Utskrift:

```
Some sound
Fetching stick!
Some sound
Some sound
```

Legg for øvrig merke til at i denne implementasjonen har ikke `Dog` og `Cat` sin egem implementasjon av `speak()`, så `Animal` sin `speak()` brukes (`Animal` er ikke abstrakt her…).

© Frode Næsje

10.8 Account utvidet til et klassehierarki med subklasser

Vi bruker det vi har lært, og utvider Account klassen til et klassehierarki samt en Bank klasse som holder oversikt over kontoene som er opprettet. De to barneklassene vi lager heter `SavingsAccount` og `StudentAccount`. `Account` klassen gjøres om til en abstrakt klasse; - den er altså ikke mulig å instansiere.

Koden, med utskrift og grovmasket forklaring:

```
# file: sc_10_05_Account5_w_subclasses1.py
 1: from datetime import datetime
 2: from abc import ABC, abstractmethod
 3: from datetime import datetime
 4:
 5: class Transaction:
 6:     def __init__(self, amount, trans_type):
 7:         self._amount = amount
 8:         self._trans_type = trans_type
 9:         self._timestamp = datetime.now()
10:     def __str__(self):
11:         return f"{self._timestamp:%Y-%m-
%d %H:%M:%S} : {self._trans_type.capitalize():8} : {self._amount:
8.2f}"
12:
13: class Account(ABC):
14:     def __init__(self, cust_id, start_balance, interest):
15:         self._cust_id = cust_id
16:         self._balance = start_balance
17:         self._interest = interest
18:         self._transactions = []
19:
20:     @abstractmethod
21:     def account_type(self):
22:         pass
23:
24:     def deposit(self, amount):
25:         if amount > 0:
26:             self._balance += amount
27:             self._transactions.append(Transaction(amount, "de
posit"))
28:         return self._balance
29:
30:     def add_monthly_interest(self):
31:         monthly_interest = self.calculate_monthly_interest()
```

© Frode Næsje

```
32:         self._balance += monthly_interest
33:         self._transactions.append(Transaction(monthly_interes
t, "interest"))
34:
35:     def calculate_monthly_interest(self):
36:         return self._balance * self._interest / 100 / 12
37:
38:     def withdraw(self, amount):
39:         if amount <= self._balance:
40:             self._balance -= amount
41:             self._transactions.append(Transaction(amount, "wi
thdraw"))
42:         return self._balance
43:
44:     @property
45:     def balance(self):
46:         return self._balance
47:
48:     def __str__(self):
49:         return f"Customer id  = {self._cust_id}\nBalance
 = {self._balance:.2f}\nInterest     = {self._interest}%\n"
50:
51: class SavingsAccount(Account):
52:     def __init__(self, cust_id, start_balance, interest, savi
ngs_goal=0):
53:         super().__init__(cust_id, start_balance, interest)
54:         self._savings_goal = savings_goal
55:
56:     def add_monthly_interest(self):
57:         # Savings accounts get double interest for demonstrat
ion
58:         monthly_interest = self.calculate_monthly_interest()
* 2
59:         self._balance += monthly_interest
60:         self._transactions.append(Transaction(monthly_interes
t, "interest"))
61:
62:     def account_type(self):
63:         return "SavingsAccount"
64:
65: class StudentAccount(Account):
66:     def __init__(self, cust_id, start_balance, interest, stud
ent_id=None):
67:         super().__init__(cust_id, start_balance, interest)
68:         self._student_id = student_id
69:
70:     def withdraw(self, amount):
```

© Frode Næsje

```
71:         # Student accounts cannot withdraw more than 1000 at
a time
72:         if amount > 1000:
73:             print("StudentAccount: Cannot withdraw more than
1000 at a time!")
74:             return self._balance
75:         return super().withdraw(amount)
76:
77:     def account_type(self):
78:         return "StudentAccount"
79:
80: class Bank:
81:     def __init__(self):
82:         self._accounts = []
83:         self._savings_count = 0
84:         self._student_count = 0
85:
86:     def add_account(self, account):
87:         self._accounts.append(account)
88:         if isinstance(account, SavingsAccount):
89:             self._savings_count += 1
90:         elif isinstance(account, StudentAccount):
91:             self._student_count += 1
92:
93:     def print_account_summary(self):
94:         print(f"Totalt: {len(self._accounts)} kontoer")
95:         print(f"  SavingsAccount: {self._savings_count}")
96:         print(f"  StudentAccount: {self._student_count}")
97:
98:     def do_monthly_update(self):
99:         print("\nMånedlig oppdatering:")
100:          for acc in self._accounts:
101:              acc.add_monthly_interest()  # Polymorft kall
102:              print(f"{acc.account_type()} etter rente: {acc.b
alance:.2f}")
103:
```

Klientkode:

```
if __name__ == "__main__":
    bank = Bank()
    # acc1 = Account("A123", 5000, 2.0)  # Kan ikke lenger
instansieres direkte
    acc2 = SavingsAccount("B456", 2000, 1.5, savings_goal=10000)
    acc3 = StudentAccount("C789", 1500, 1.2, student_id="STU123")
    bank.add_account(acc2)
    bank.add_account(acc3)

    bank.print_account_summary()
```

© Frode Næsje

```
    # Demonstrer polymorfi og isinstance
    for acc in bank._accounts:
        print(f"\nType: {acc.account_type()}")
        acc.deposit(500)
        acc.add_monthly_interest()
        acc.withdraw(1200)
        print(acc)
        # Bruk av isinstance:
        if isinstance(acc, SavingsAccount):
            print(f"  Sparemål: {acc._savings_goal}")
        if isinstance(acc, StudentAccount):
            print(f"  Student-ID: {acc._student_id}")

    bank.do_monthly_update()
```

Utskrift:

```
Totalt: 2 kontoer
  SavingsAccount: 1
  StudentAccount: 1

Type: SavingsAccount
Customer id  = B456
Balance      = 1306.25
Interest     = 1.5%

  Sparemål: 10000

Type: StudentAccount
StudentAccount: Cannot withdraw more than 1000 at a time!
Customer id  = C789
Balance      = 2002.00
Interest     = 1.2%

  Student-ID: STU123

Månedlig oppdatering:
SavingsAccount etter rente: 1309.52
StudentAccount etter rente: 2004.00
```

Forklaring til kode
Linje 5 – 12:

© Frode Næsje

Transaction klassen er ikke endret

Linje 13 – 50
`Account` klassen er nå abstrakt og kan ikke instansieres. `Account` har en blanding av abstrakte og implementerte metoder:

- `account_type(self)` er abstrakt og må implementeres i barneklassene
- `deposit(self, amount)` er kun implementer her, og overrides ikke i noen av barneklassene
- `add_monthly_interest(self)` er implementer her og brukes av `StudentAccount`, men ikke av `SavingsAccount`, som har en annen måte å beregne rente på
- `calculate_monthly_interest(self)` er implementert her og brukes av både `StudentAccount` og `SavingsAccount`
- `withdraw(self, amount)` brukes av `SavingsAccount`, men ikke av `StudentAccount`, som ønsker å implementere sin egen versjon av `withdraw()`

Linje 51 – 63
I linje 53 kalles constructor i `Account` klassen, hvor nødvendige data som `Account` klassen har ansvaret for blir videresendt. I linje 54 tas vare på `SavingsAccount` sin egen attributt `savings_goal`.

I linje 56 – 60 implementeres en egen versjon av `add_monthly_interest(self)`, som bruker `Account` sin implementasjon av `calculate_monthly_interest()` for å beregne rente, for deretter å multiplisere med 2.

I Linje 62 – 63 implementeres `account_type(self)`, som bare returnerer en tekst som sier hvilken type konto dette er. Dette informasjonen kunne vi også fått tak i på annen måte: `type()` og `isinstance()`, men `account_type()` gir konto typen en god, tekstlig beskrivelse.

Linje 65 - 78
Implementasjonen av `StudentAccount`, som har en egen implementasjon av `withdraw(self,amount)` og `account_type(self)`.

© Frode Næsje

I linje 75 utfører vi kallet `super. withdraw(amount);` - hadde vi kalt funksjonen slik: `self.withdraw(amount)`, så ville vi kalt `StudentAccount` sin versjon, og vi hadde fått en evig loop (rekursivt kall til seg selv)

I linje 77 – 78 har vi implementasjonen av `StudentAccount` sin `account_type()` se beskrivelse ovenfor av samme for `SavingsAccount`

Linje 80 – 102
Implementasjon av `Bank` klassen, som har en liste over kontoer opprettet, samt to tellere for de to konto typene vi har. Disse tellerne er vanlige attributter i bank klassen, og vi teller de opp i metoden `add_account()` etter at vi har sjekket med `isinstance()` hvilken type objekt det er vi legger til.

I linje 98 – 102 gjør vi en månedlig beregning av rente, og kallet `acc.add_monthly_interest()` vil enten gå til `Account` eller `SavingsAccount` sin versjon av denne metoden, se tidligere forklaring.

Klientkode:
Se at koden samsvarer med utskrift

© 2025 [Frode Næsje]
Alle rettigheter reservert. Ingen deler av denne publikasjonen kan reproduseres, distribueres, eller overføres i noen form eller på noen måte, elektronisk, mekanisk, fotokopiering, opptak, eller på annen måte, uten forhåndstillatelse fra forfatteren.

UTKAST