Objektorientiertes Programmieren III

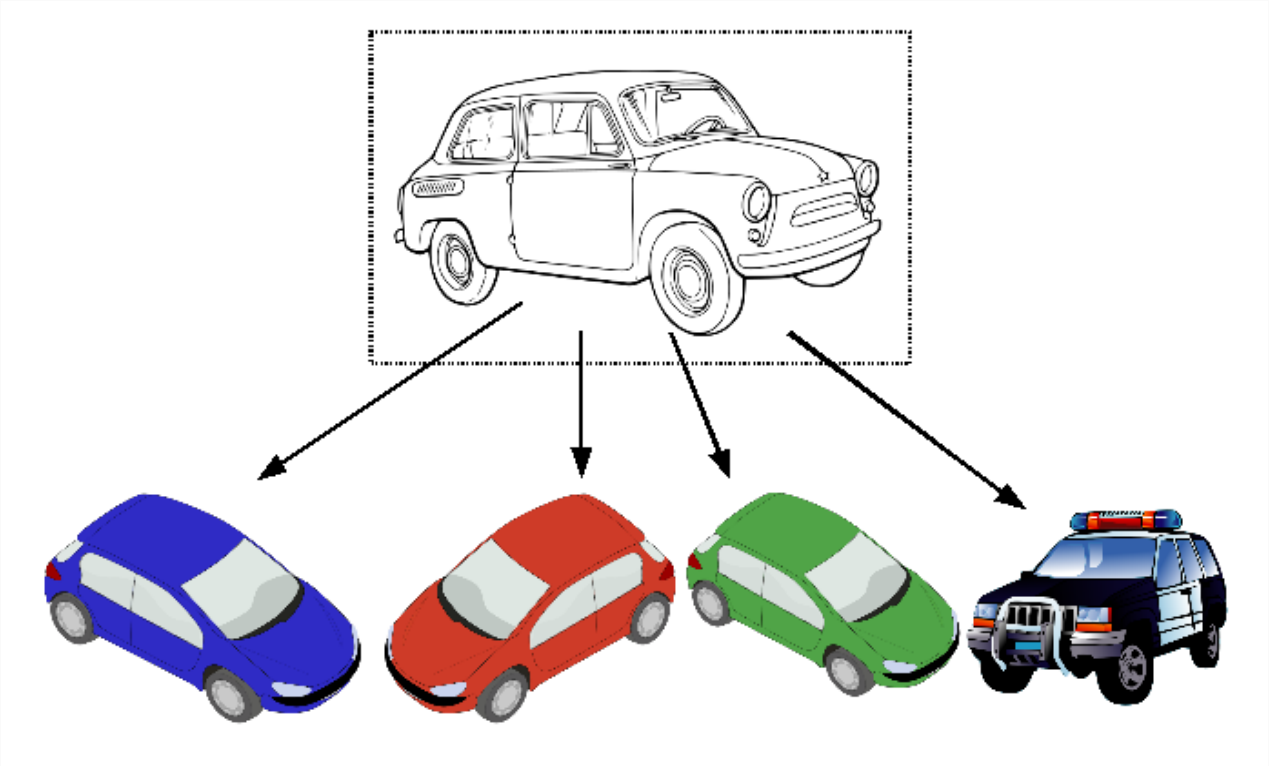
Symbolische Programmiersprache

Benjamin Roth and Annemarie Friedrich

Wintersemester 2016/2017

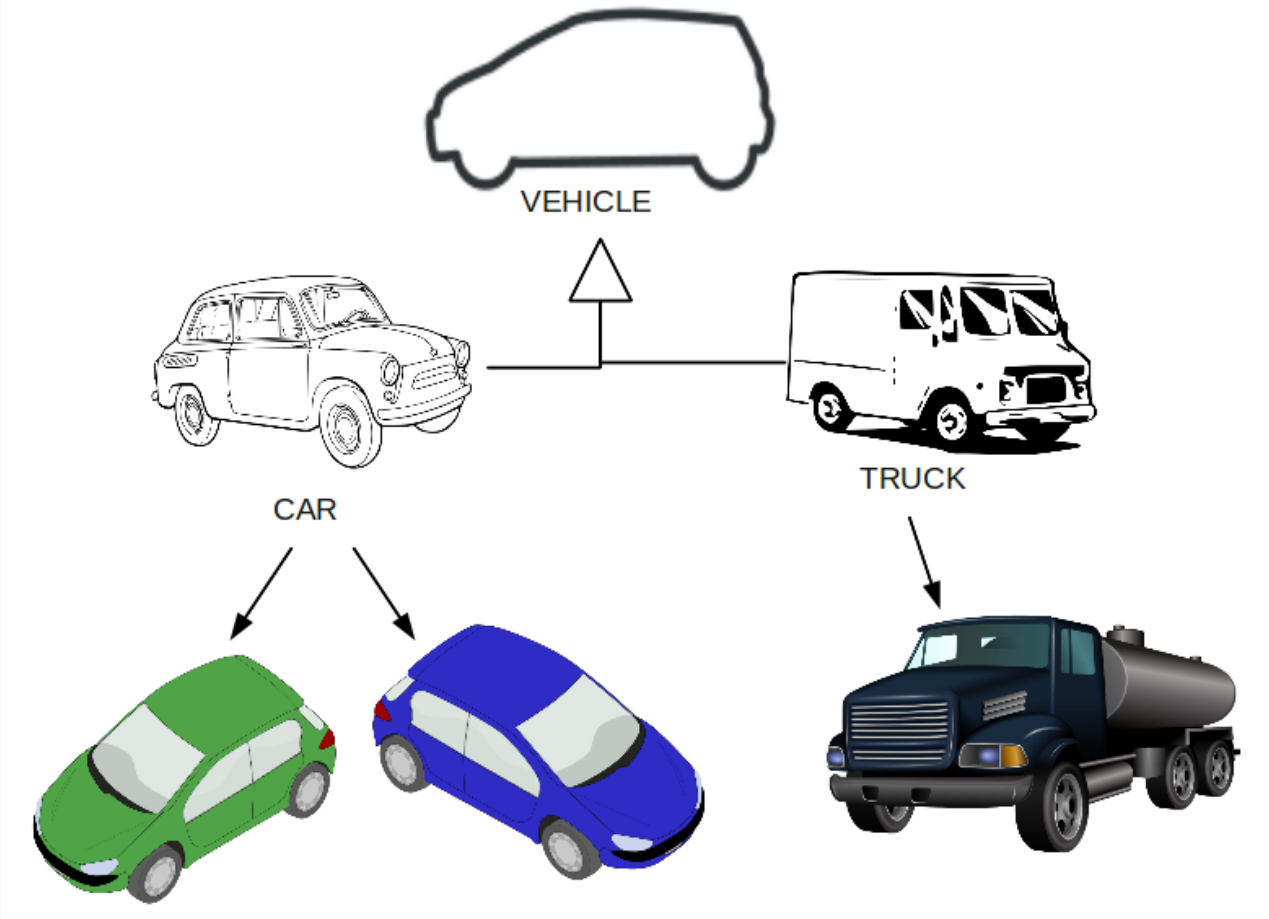
Centrum für Informations- und Sprachverarbeitung
LMU München

Recap: Klassen = Baupläne



- Klassen sind Baupläne/Designs für Objekte.
- Objekte mit Hilfe von Klassen erstellen: Wir instanziieren objects. Objekte = Instanzen einer Klasse.
- Objekte einer Klasse haben die gleiche grundlegende Struktur, können sich aber in verschiedenen Aspekten unterscheiden.

Recap: Vererbung



- Verschiedene Objekte haben teilweise dasselbe Verhalten / dieselben Charakteristika.
- Vererbung → Code-Dopplungen vermeiden.

Beispiel Bank

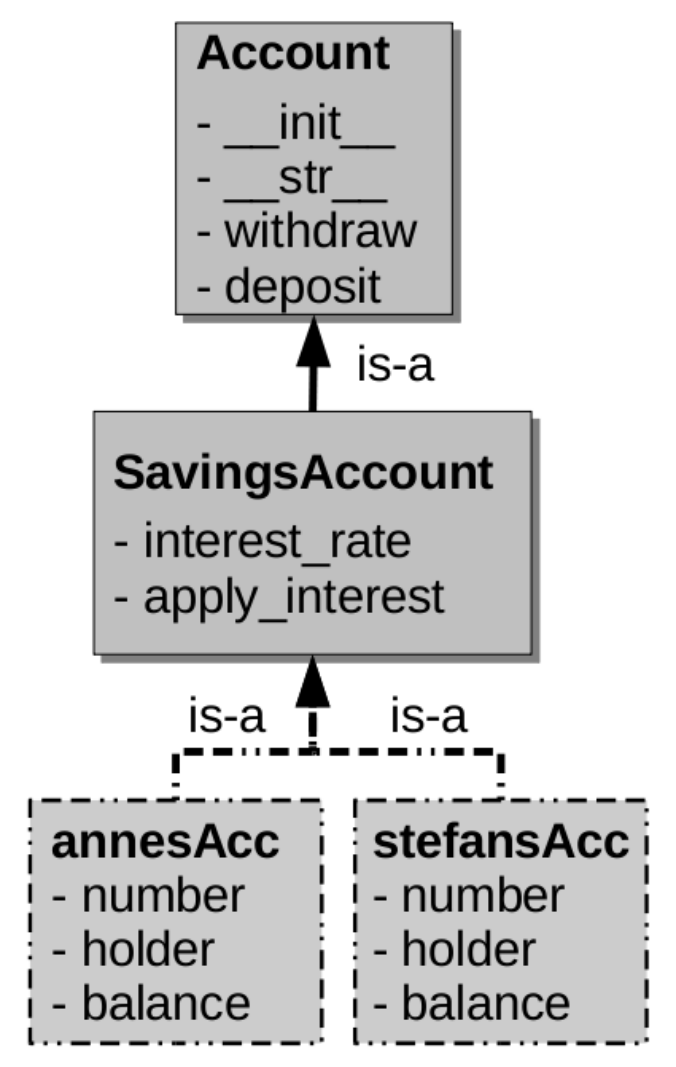
- **Sparkonto**: Für jedes Konto werden Kontonummer, Kontoinhaber und Kontostand gespeichert. Der Kontostand muss ≥ 0 sein. Eine Zinsrate, die für alle Sparkonten gemeinsam definiert ist, kann angewendet werden. Auf das Konto kann Geld eingezahlt werden. Der Kontoauszug (der ausgedruckt werden kann) beinhaltet die Kontonummer, den Kontoinhaber und den Kontostand.
- **Girokonto**:Für jedes Konto werden Kontonummer, Kontoinhaber und Kontostand gespeichert. Der Kontostand muss innerhalb dem für jeden Kunden separat definierten Kreditrahmen liegen. Wenn der Kreditrahmen von Anne $500 ist, darf ihr Kontostand minimal -$500 betragen. Auf das Konto kann Geld eingezahlt werden. Der Kontoauszug (der ausgedruckt werden kann) beinhaltet die Kontonummer, den Kontoinhaber und den Kontostand.

Klassendesign: Tafel

Gemeinsamkeiten/Generelle Funktionalität: Überklasse

```
1  class Account:
2      ''' a class providing general
3      functionality for accounts  '''
4      # CONSTRUCTOR
5      def __init__(self, num, person):
6          self.balance = 0
7          self.number = num
8          self.holder = person
9      # METHODS
10     def deposit(self, amount):
11         self.balance += amount
12     def withdraw(self, amount):
13         if amount > self.balance:
14             amount = self.balance
15         self.balance -= amount
16         return amount
17     def __str__(self):
18         res = ...
19         return res
```

Spezialfälle: Unterklassen



- `SavingsAccount` “basiert auf” `Account`
- Methoden der Überklasse sind in der Unterklasse (und Instanzobjekten der Unterklasse) verfügbar

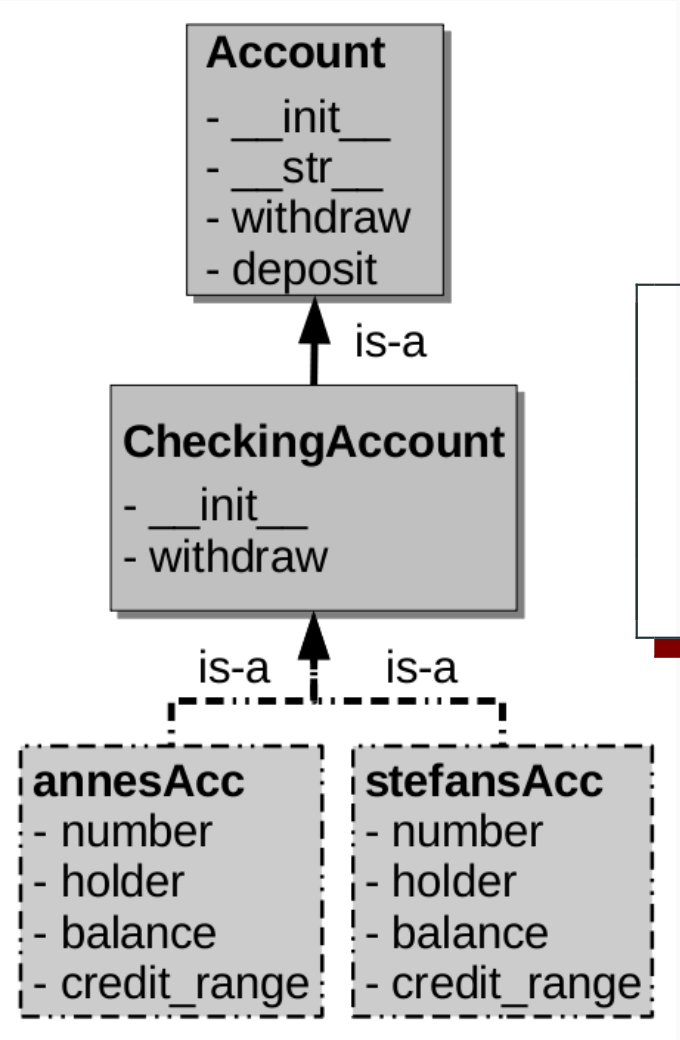
```
1  annesAcc = SavingsAccount(1, "Anne")
2  annesAcc.deposit(200)
3  annesAcc.apply_interest()
4  print(annesAcc)
```

Spezialfälle: Unterklassen

- `SavingsAccount` is based on `Account`
- `SavingsAccount` is derived from `Account`
- `SavingsAccount` extends `Account`
- Methoden der Überklasse sind in der Unterklasse (und Instanzobjekten der Unterklasse) verfügbar
- `SavingsAccount` stellt zusätzliche Funktionalität bereit

```
1  class SavingsAccount(Account):
2      ''' class for objects representing savings accounts.
3          shows how a class can be extended. '''
4      interest_rate = 0.035
5      # METHODS
6      def apply_interest(self):
7          self.balance *= (1+SavingsAccount.interest_rate)
```

Methoden überschreiben



- Die `Account`-Klasse hat auch `__init__` and `withdraw` Methoden
- Die `CheckingAccount`-Klasse überschreibt diese

```
1  annesAcc = CheckingAccount(1,
2                 "Anne", 500)
3  annesAcc.deposit(200)
4  annesAcc.withdraw(350)
5  print(annesAcc)
```

Welche Methoden werden aufgerufen?

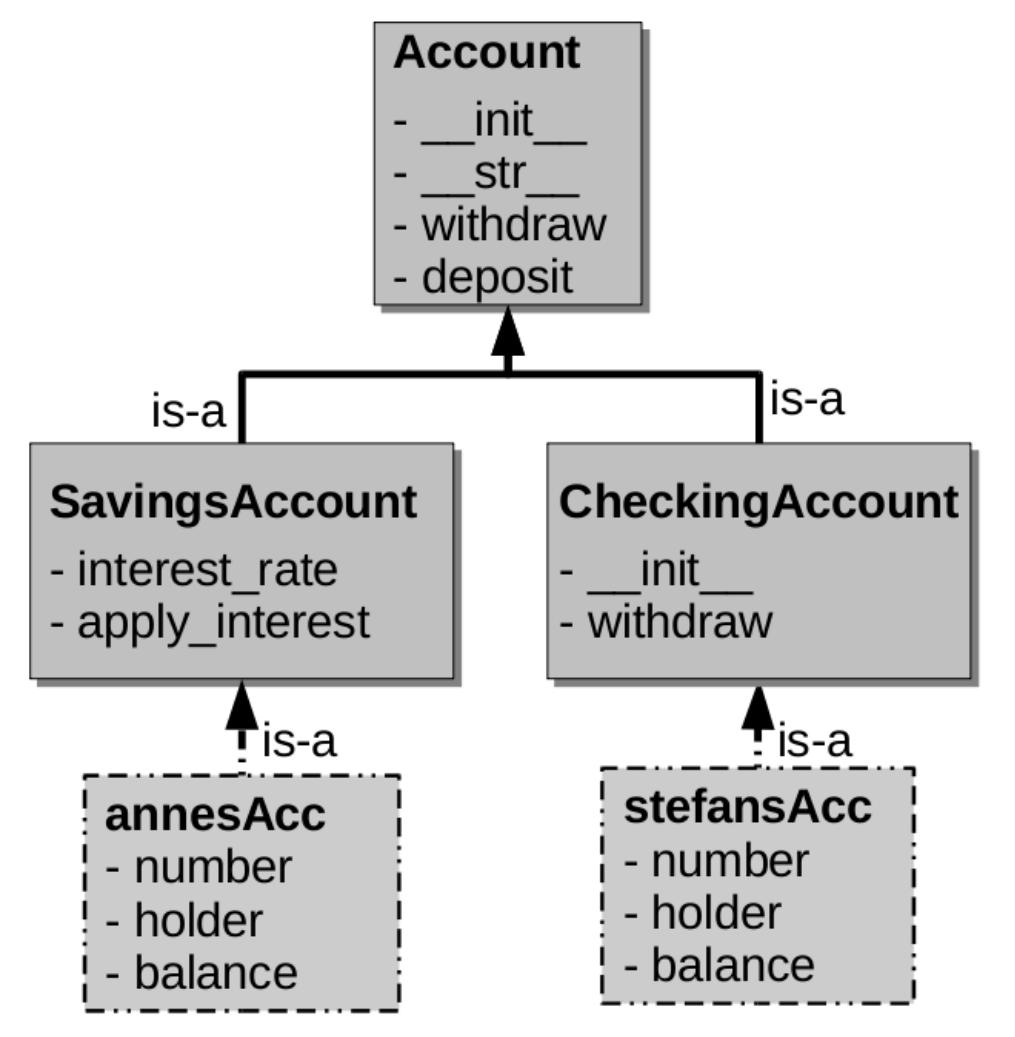
- `__init__`
- `deposit`
- `withdraw`
- `__str__`

Methoden überschreiben

- Die `Account`-Klasse hat auch `__init__` and `withdraw` Methoden
- Die `CheckingAccount`-Klasse überschreibt diese

```
1  class CheckingAccount(Account):
2      # CONSTRUCTOR
3      def __init__(self, num, person, credit_range):
4          print("Creating a checkings account")
5          self.number = num
6          self.holder = person
7          self.balance = 0
8          self.credit_range = credit_range
9      # METHODS
10     def withdraw(self, amount):
11         amount = min(amount, abs(self.balance \
12             + self.credit_range))
13         self.balance -= amount
14         return amount
```

Klassenhierarchie



Welche Methoden werden ausgeführt, wenn die folgenden Methoden auf `annesAcc` oder `stefansAcc` aufgerufen werden?

- `__init__`
- `__str__`
- `deposit`
- `withdraw`
- `apply_interest`

Polymorphismus

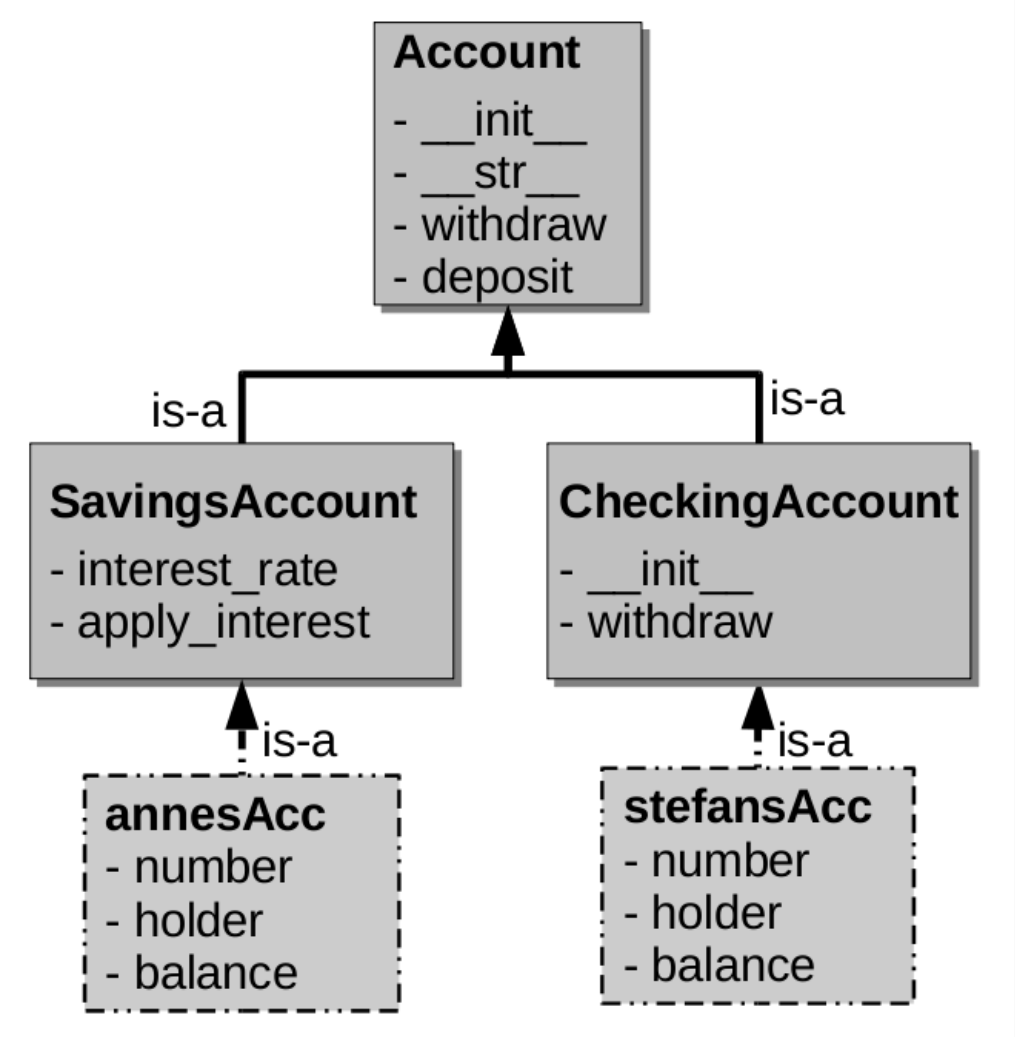
- 'Vielgestaltigkeit' (Griechisch)
- Methode wird auf Objekte angewandt $\rightarrow$ was passiert, hängt von der Klassenhierarchie ab

Beispiel

Anwendungscode: `annesAcc.withdraw(400)`

- Es ist egal, von welchem Typ `annesAcc` ist
- Python folgt der Klassenhierarchie und erzeugt das gewünschte Verhalten

Klassenhierarchie: Design



Wir hätten auch zwei `withdraw`-Methoden definieren können (eine in jeder Unterklasse).
Warum könnte es nützlich sein, sie in der `Account`-Klasse zu haben?

Redundanz

- Wenn die Daten für ein Objekt mehrfach existieren (z.B. Kontoinhaber-Info separat für jedes Konto führen)
  $\Rightarrow$ Inkonsistenzen möglich
- Derselbe Code wird mehrfach geschrieben
  $\Rightarrow$ Wartung schwierig

```
1  class Account:
2      def __init__(self, num, person):
3          self.balance = 0
4          self.number = num
5          self.holder = person
6
7  class CheckingAccount(Account):
8      def __init__(self, num, person, credit_range):
9          self.number = num
10         self.holder = person
11         self.balance = 0
12         self.credit_range = credit_range
```

Redundanzen minimieren

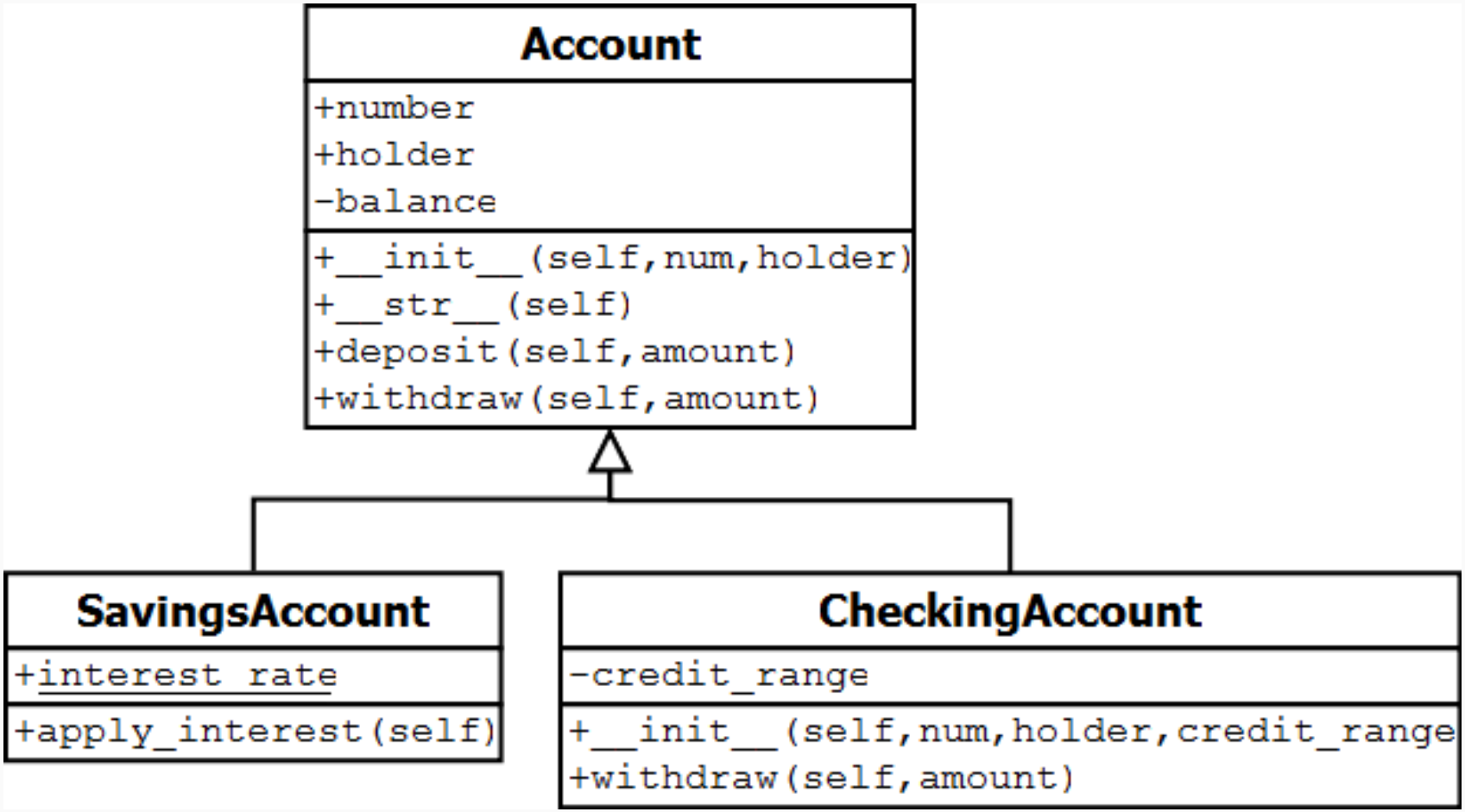
- Hier: Methode einer Überklasse aus einer Unterklasse aufrufen
  $\Rightarrow$ Methode der Überklasse erweitern
- Methode wird *auf der Klasse* aufgerufen $\Rightarrow$ Referenz auf Instanzobjekt muss hier explizit an `self` übergeben werden

```
1  class Account:
2      def __init__(self, num, person):
3          self.balance = 0
4          self.number = num
5          self.holder = person
6
7  class CheckingAccount(Account):
8      def __init__(self, num, person, credit_range):
9          Account.__init__(self, num, person)
10         self.credit_range = credit_range
```

Redundanzen minimieren: weiteres Beispiel

```
1  class Account:
2      def withdraw(self, amount):
3          self.balance -= amount
4          return amount
5
6  class SavingsAccount(Account):
7      def withdraw(self, amount):
8          if amount > self.balance:
9              amount = self.balance
10         cash = Account.withdraw(self, amount)
11         return cash
12
13 class CheckingAccount(Account):
14     # METHODS
15     def withdraw(self, amount):
16         amount = min(amount,
17                  abs(self.balance + self.credit_range))
18         cash = Account.withdraw(self, amount)
19         return cash
```

UML Klassendiagramm: Vererbung



Mehrfachvererbung

- In einigen objektorientierten Sprachen (z.B. Java) können Klassen nur eine einzige Klasse erweitern.
- In Python kann eine Klasse von mehreren Klassen erben
  $\Rightarrow$ Multiple Inheritance (Mehrfachvererbung)
- Methodenaufruf $\rightarrow$ welche Klasse? $\rightarrow$ spezielle Mechanismen
- Empfehlung: im Moment max. eine Überklasse nutzen!

Everything in Python is an object

- Wir benutzen eigentlich die ganze Zeit schon Objekte ...
- Listen und Dictionaries sind Objekt
- Erzeugung mit spezieller Syntax (Klassenaufruf auch möglich)
- Strings und Zahlen sind auch Objekte

```
1  # create a new list object
2  myList = []
3  # call a method of the list object
4  myList.append(4)
5  # create a new dictionary object
6  myDict = {}
7  # call a method of the dictionary object
8  myDict["someKey"] = "someValue"
```

- Zeile 8 ruft eine Hook-Methode des Dictionaries auf:

  `__setitem__(self, key, value)`

Everything in Python is an object

- Wir können auch Unterklassen der **built-in**-Klassen erstellen
- Hier: Hooks überschreiben

```
1  class TalkingDict(dict):
2      # Constructor
3      def __init__(self):
4          print("Starting to create a new dictionary...")
5          dict.__init__(self)
6          print("Done!")
7      # Methods
8      def __setitem__(self, key, value):
9          print("Setting", key, "to", value)
10         dict.__setitem__(self, key, value)
11         print("Done!")
12
13 print("We are going to create a talking dictionary!")
14 myDict = TalkingDict()
15 myDict["x"] = 42
```

Verständnis von OOP ist nützlich, weil ...

- Wir programmieren “im wahren Leben” selten *from scratch*
- Wir erweitern / modifizieren bestehenden Code
- Framework = Sammlung von (Über-)Klassen, die häufige Programmieraufgaben implementieren
- Wir müssen verstehen, wie die Klassen des Frameworks zusammenarbeiten
- Wir schreiben Unterklassen, die das Verhalten für unseren Anwendungsfall spezialisieren
- Rezepte dafür: Design Patterns
  (*Design Patterns: Elements of Reusable Object-Oriented Software*
  by: Gamma, Helm, Johnson, Vlissides)

Terminologie: Datenkapselung (encapsulation)

- Wird in der OOP-Literatur für zwei Aspekte benutzt:

1. Daten sollten *versteckt* sein, Zugriff nur über Instanzmethoden
2. Programm-Logik soll in *Schnittstellen* (Interfaces) (Klassennamen und Methodennamen) so verpackt sein, dass jede Funktionalität nur einmal definiert ist

References


Mark Lutz: *Learning Python*, Part VI, 4th edition, O'Reilly, 2009.


Michael Dawson: *Python Programming for the Absolute Beginner*, Chapters 8 & 9, 3rd edition, Course Technology PTR, 2010.