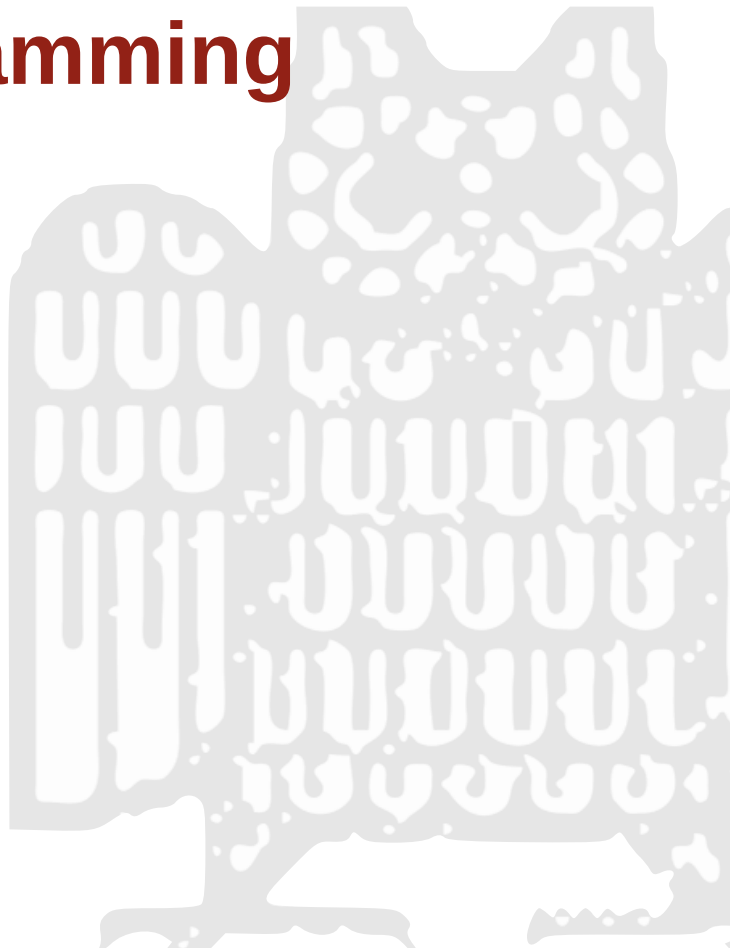
# Introduction to Python Programming

## 14 – Object Orientation III

Josef van Genabith (Stefan Thater)
Dept. of Language Science & Technology
Universität des Saarlandes

WS 2022/23



# Recap: Instance Objects vs. Class Objects

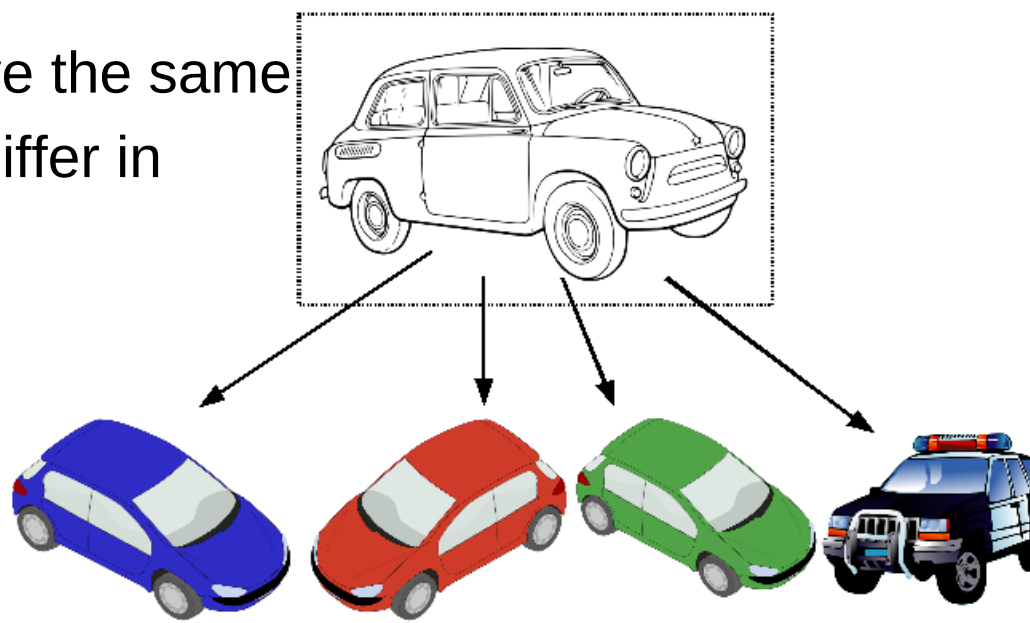
- Instance objects: instantiations of a class. Multiple instances possible.
- Instance attributes: variables of an instance object. Different values for different instances.
- Class object: exists only once per class.
- Class attributes: variables of a class object. Visible from class or instance, but always pointing to the same value.
- Static methods: method of a class object. Defined via `@staticmethod` decorator. No self attribute, no access to instance attributes and methods.

# Recap: Shadowing

- Class attributes can be viewed from class object or from instance objects, but only assigned from class.
- Assigning a class attribute value through an instance object creates a local instance attribute!
- Creating instance attributes with the same name as class attributes is called shadowing.
- Shadowing means the class attribute can not be accessed through the instance object anymore, as the locally created instance attribute blocks it.

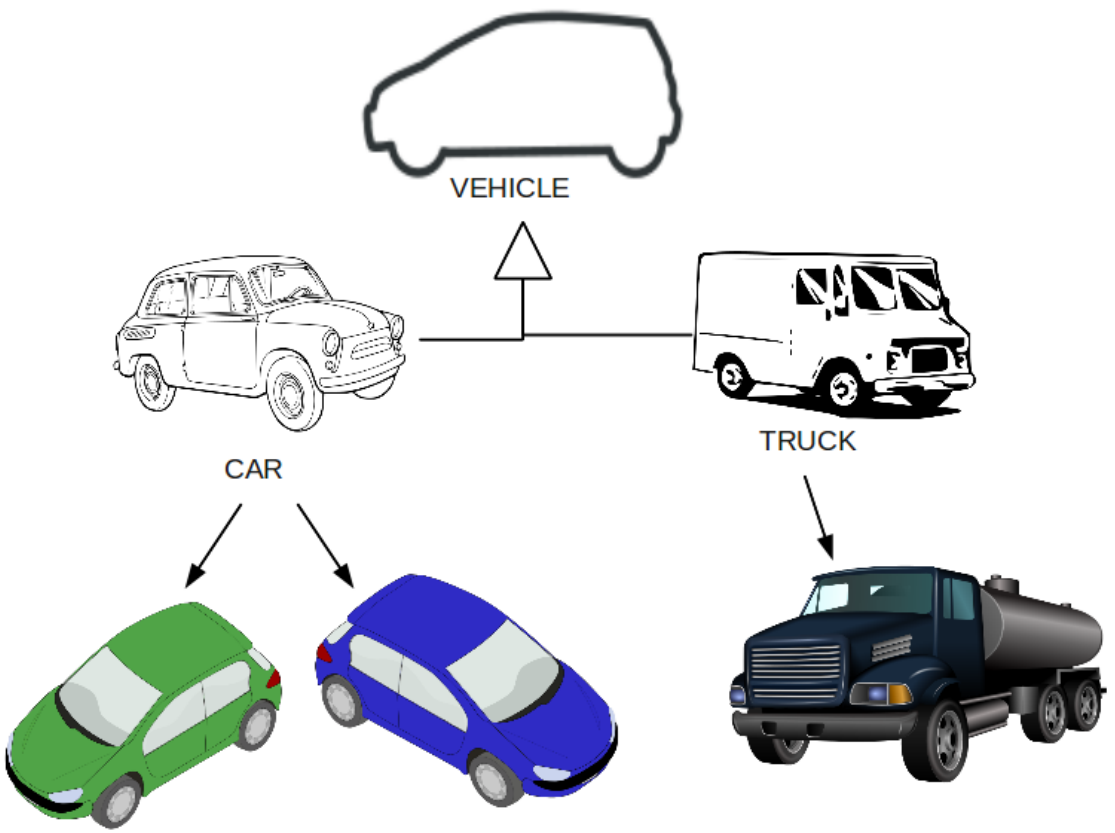
# Reminder: Classes = Blueprints

- Classes are blueprints/designs for objects.
- Creating objects using classes: we instantiate objects from a class.
- Objects are also called instances of a class.
- Objects of the same class have the same basic structure, but they can differ in various respects/attributes.



# Inheritance

- Some different objects share characteristics / behaviors.
- Inheritance saves us from writing the same code over and over again.

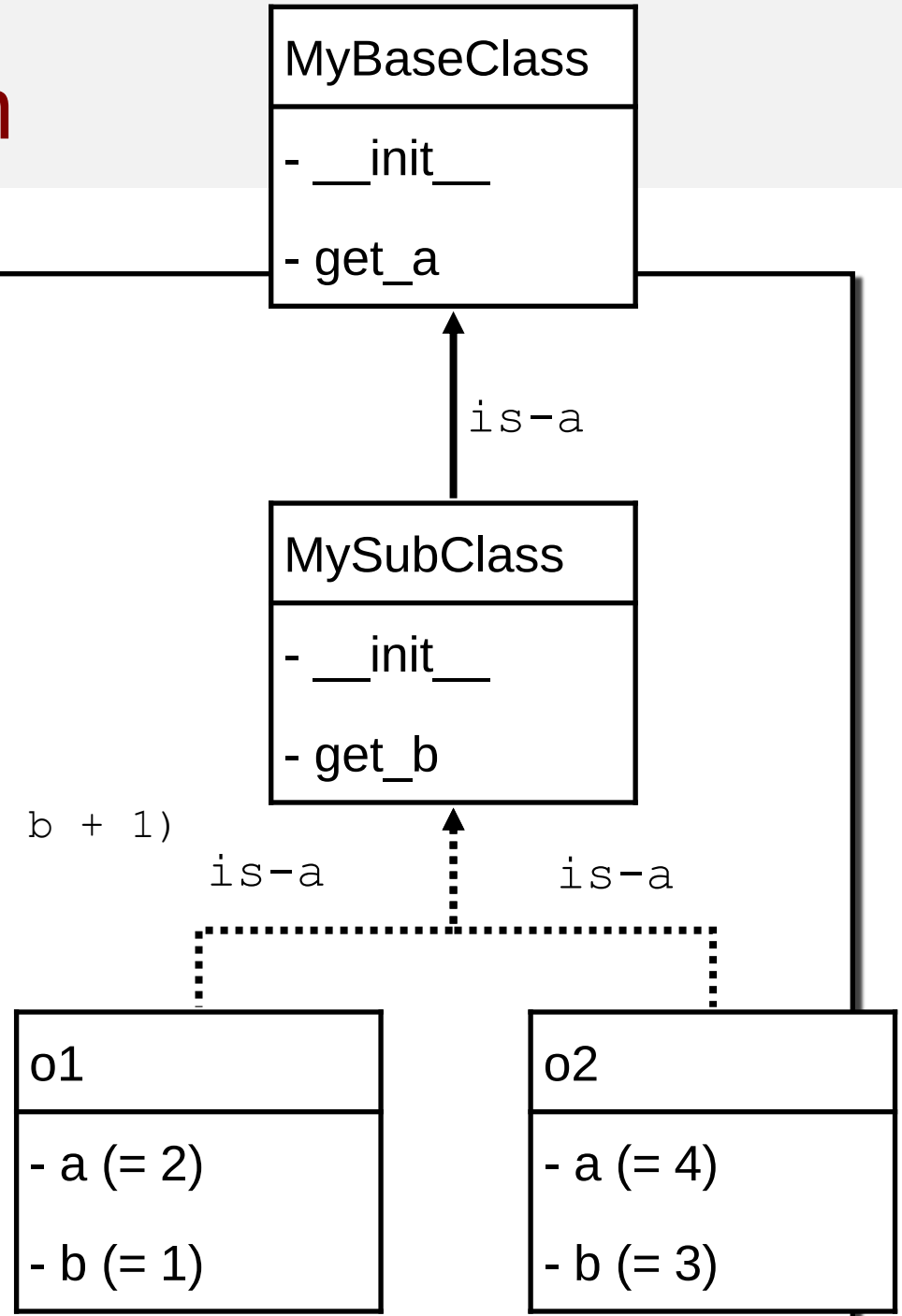


# Inheritance in Python

```
class MyBaseClass:
   def __init__(self, a):
      self.a = a
   def get_a(self):
      return self.a

class MySubClass(MyBaseClass):
   def __init__(self, b):
      MyBaseClass.__init__(self, b + 1)
      self.b = b
   def get_b(self):
      return self.b

o1 = MySubClass(1)
print(o1.get_a()) # 2
o2 = MySubClass(3)
print(o2.get_a()) # 4
```



# Example: Bank Account

**Savings Account:** We record the account number, holder and balance with each account. The balance has to be ≥ 0. We can apply an interest rate which is defined once for all savings accounts. Money can be deposited into the account. The account statement that can be printed includes the account number, holder and balance.

**Checking Account:** We record the account number, holder and balance with each account. The balance has to be greater than or equal to a credit range which is determined on a per-customer basis. Money can be deposited into the account. The account statement that can be printed includes the account number, holder and balance.

# Base Class: Commonalities

**Savings Account:** We record the account number, holder and balance with each account. The balance has to be ≥ 0. We can apply an interest rate which is defined once for all savings accounts. Money can be deposited into the account. The account statement that can be printed includes the account number, holder and balance.

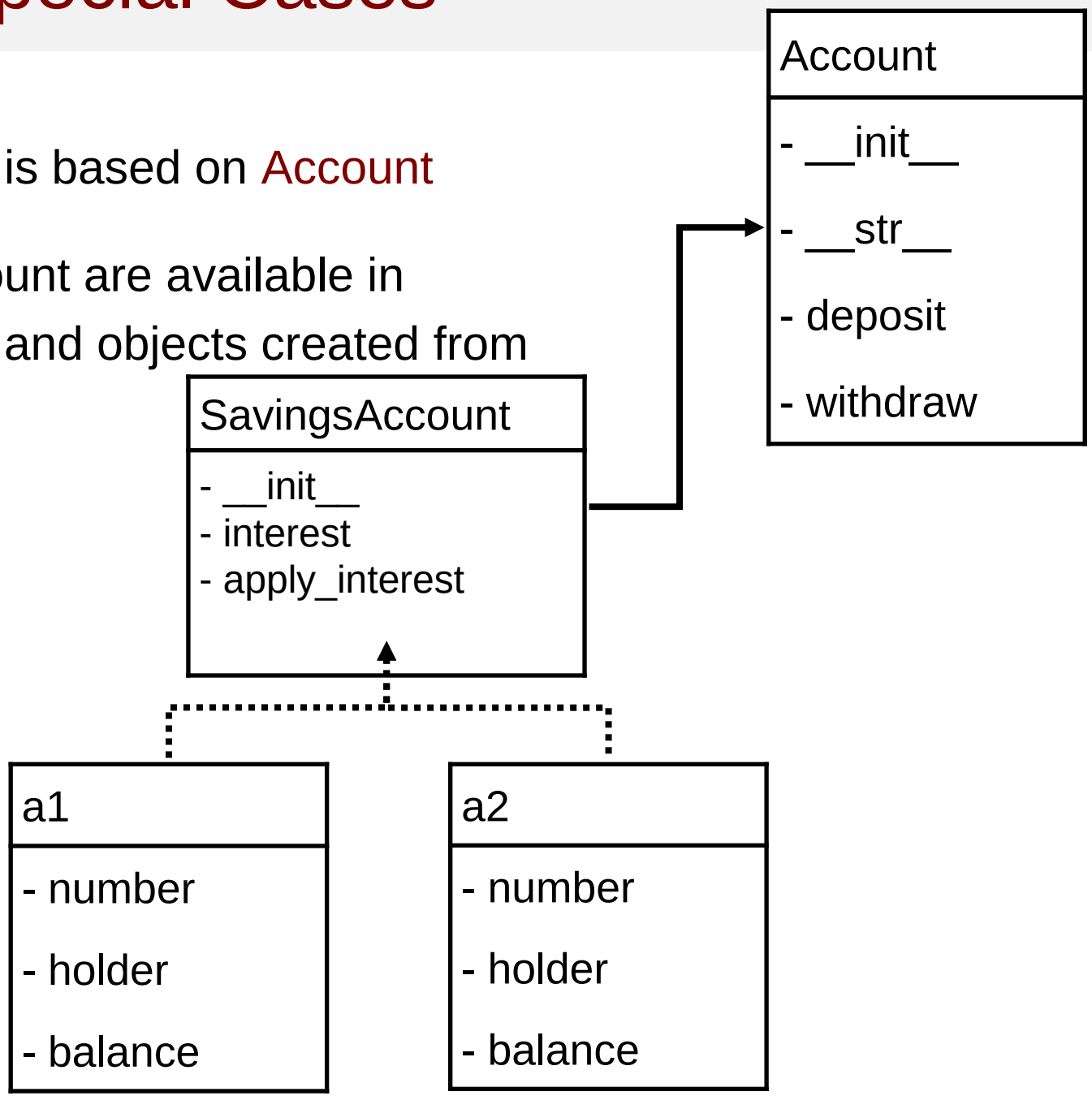
**Checking Account:** We record the account number, holder and balance with each account. The balance has to be greater than or equal to a credit range which is determined on a per-customer basis. Money can be deposited into the account. The account statement that can be printed includes the account number, holder and balance.

# Base Class: Commonalities

```
class Account:
    def __init__(self, num, person):
        self.balance = 0
        self.number = num
        self.holder = person
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        if amount > self.balance:
            amount = self.balance
        self.balance -= amount
        return amount
    def __str__(self):
        return ...
```

# Subclass: Special Cases

- SavingsAccount is based on Account
- Methods of Account are available in SavingsAccount and objects created from SavingAccount



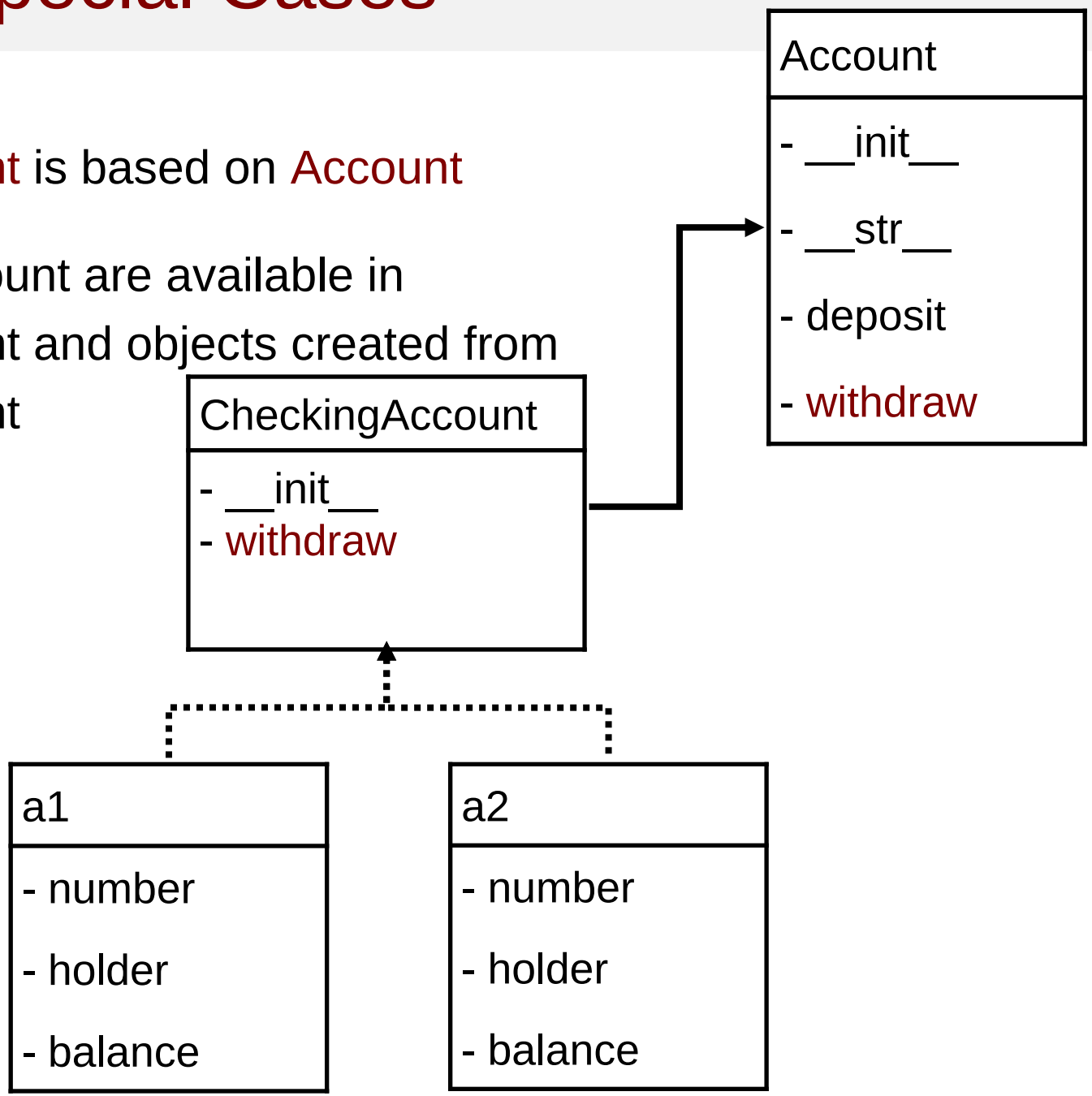
# SavingsAccount

```
class SavingsAccount(Account):
    def __init__(self, num, person, interest):
        Account.__init__(self, num, person)
        self.interest = interest       # e.g. 0.01
    def apply_interest(self):
        self.balance *= (1 + self.interest)
```

**Savings Account:** We record the account number, holder and balance with each account. The balance has to be ≥ 0. We can apply an interest rate which is defined once for all savings accounts. Money can be deposited into the account. The account statement that can be printed includes the account number, holder and balance.

# Subclass: Special Cases

- CheckingAccount is based on Account
- Methods of Account are available in CheckingAccount and objects created from CheckingAccount



# CheckingAccount

```
class CheckingAccount(Account):
    def __init__(self, num, person, credit_range):
        Account.__init__(self, num, person)
        self.credit_range = credit_range
    def withdraw(self, amount):
        amount = min(amount, abs(self.balance +
                                 self.credit_range))
        self.balance -= amount
        return amount
```

**Checking Account:** […] The balance has to be greater than or equal to a credit range which is determined on a per-customer basis. [...]

# Polymorphism

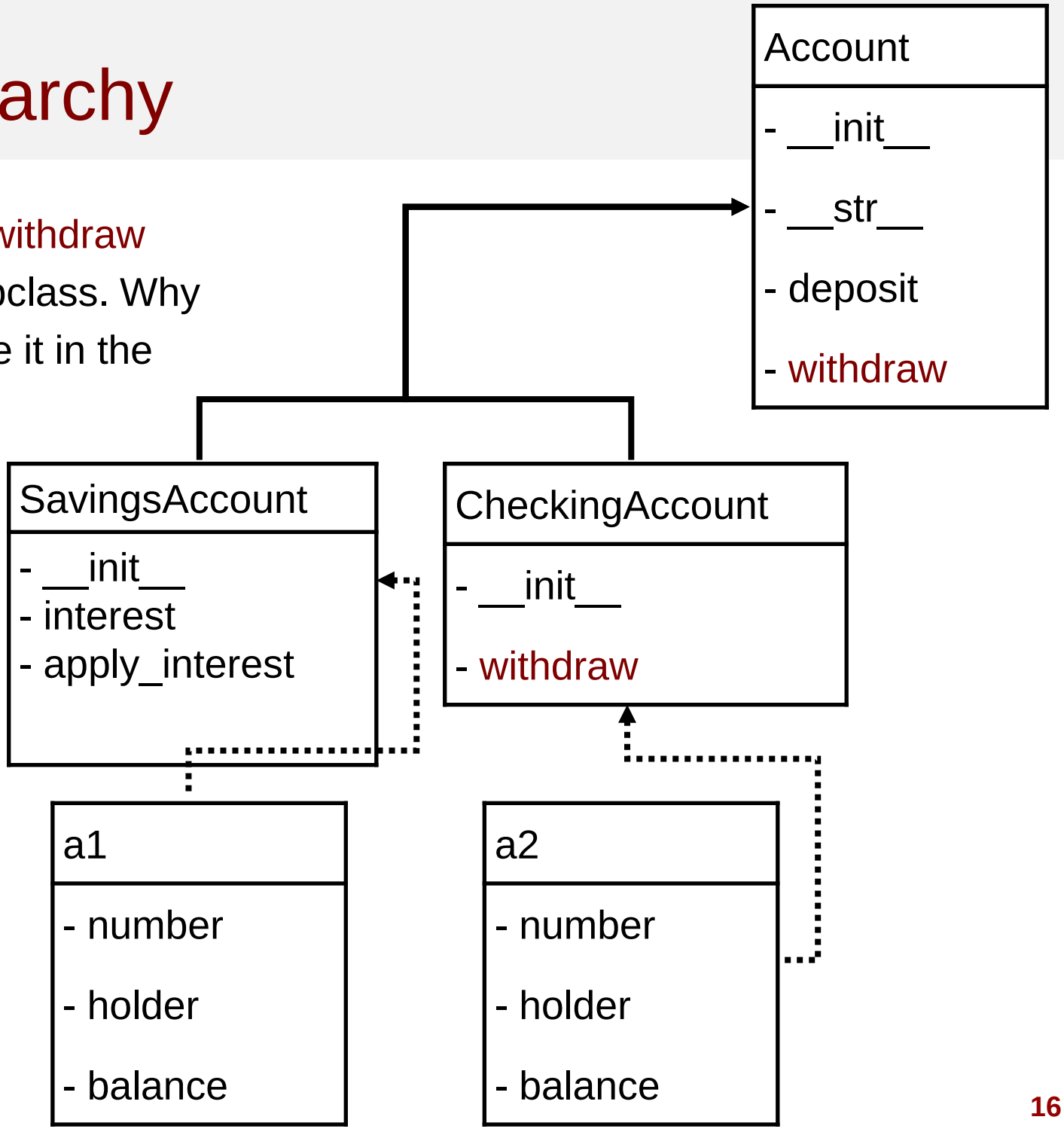
- “having multiple forms”
- When calling a method on objects, what happens depends on the class hierarchy from which the objects were created

```
>>> myAccount.withdraw(400)
```

- We don‘t need to care whether myAccount is a savings or a checking account. Python does!
- Python follows the inheritance hierarchy when looking up the withdraw method

# Class Hierarchy

We could have defined withdraw twice – once in each subclass. Why might it be useful to have it in the Account class?



# This week's exercise