Introduction to Python Programming

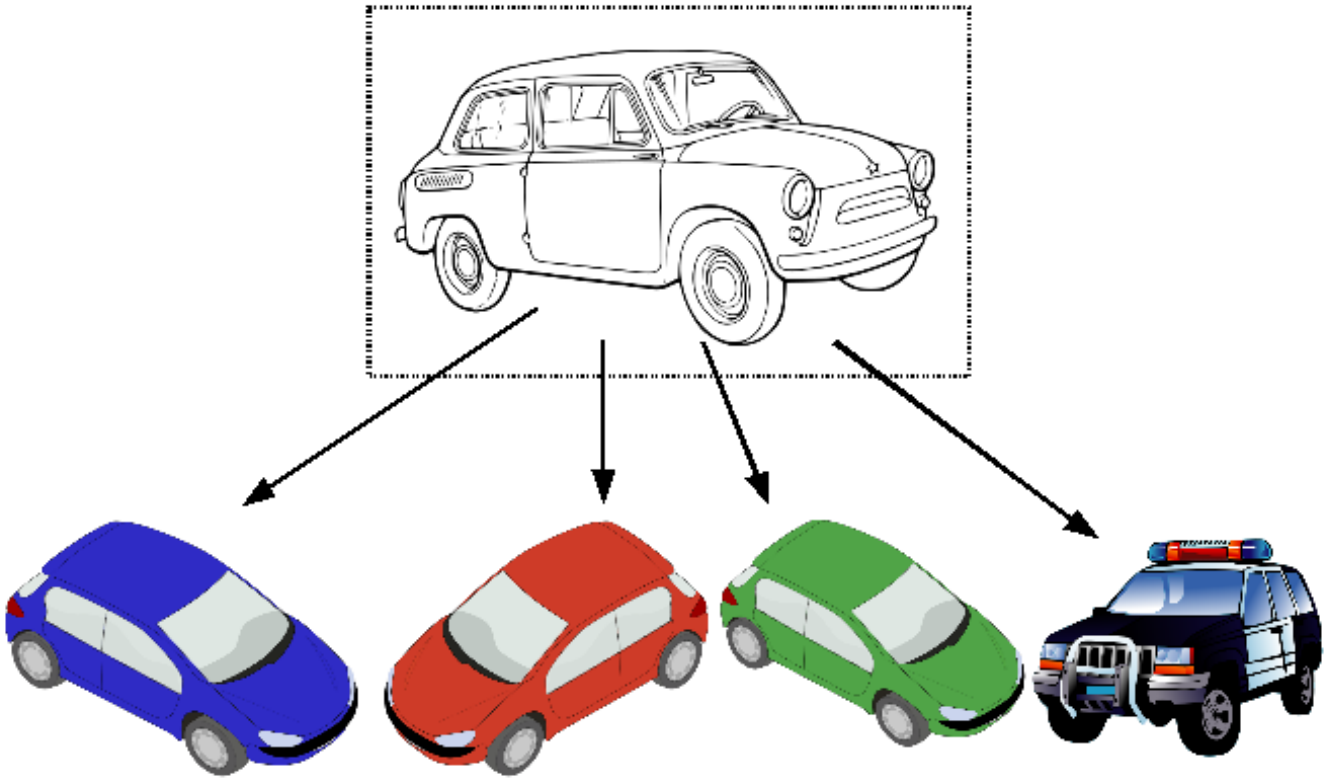
(12) Object-Oriented Programming III
Inheritance

S. Thater and A. Friedrich

Saarland University

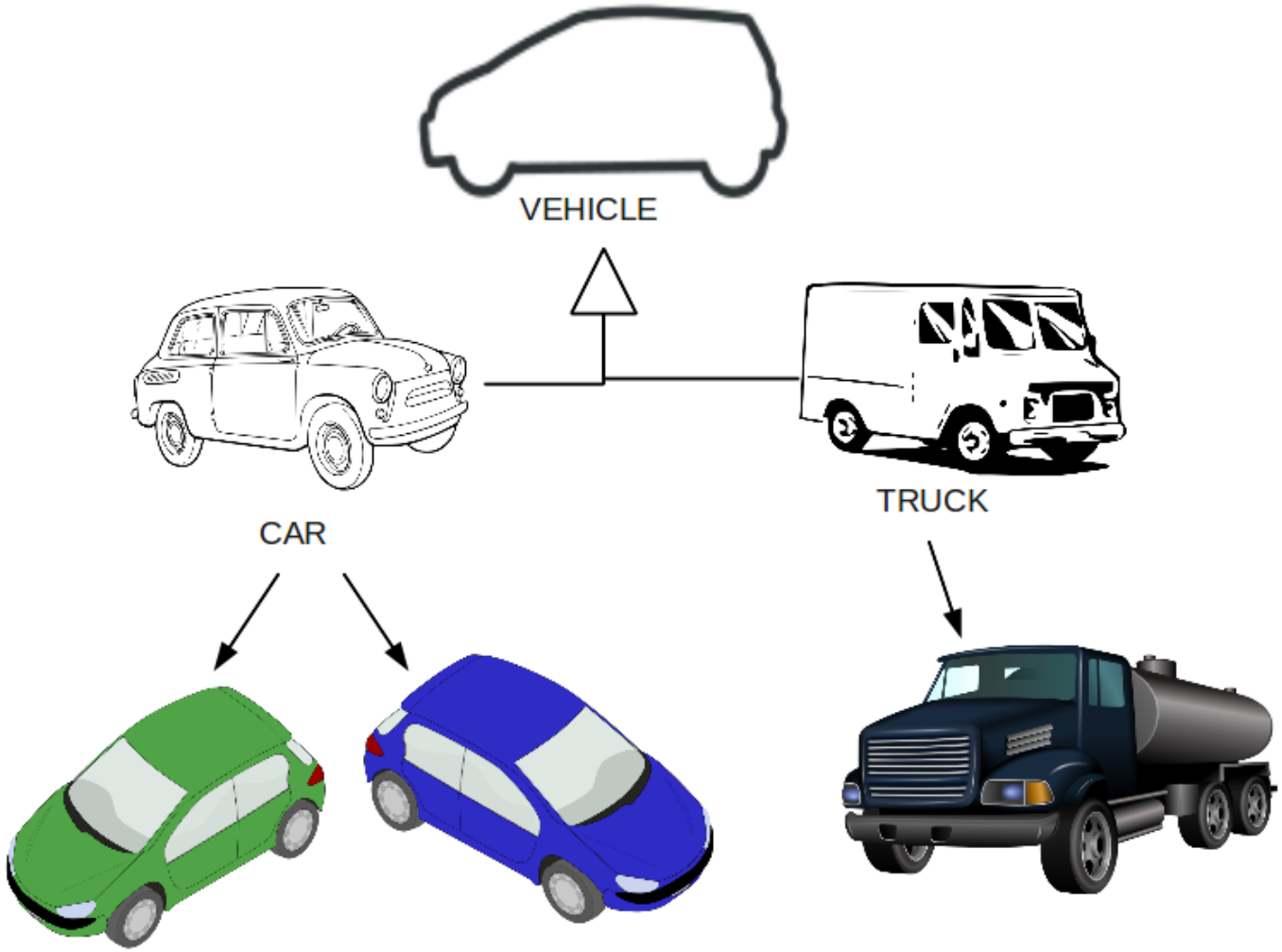
Winter Semester 2011/2012

Recap: Classes = Blueprints



- Classes are blueprints/designs for objects.
- Creating objects using classes: We instantiate objects.
  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.

Inheritance



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

Bank Example

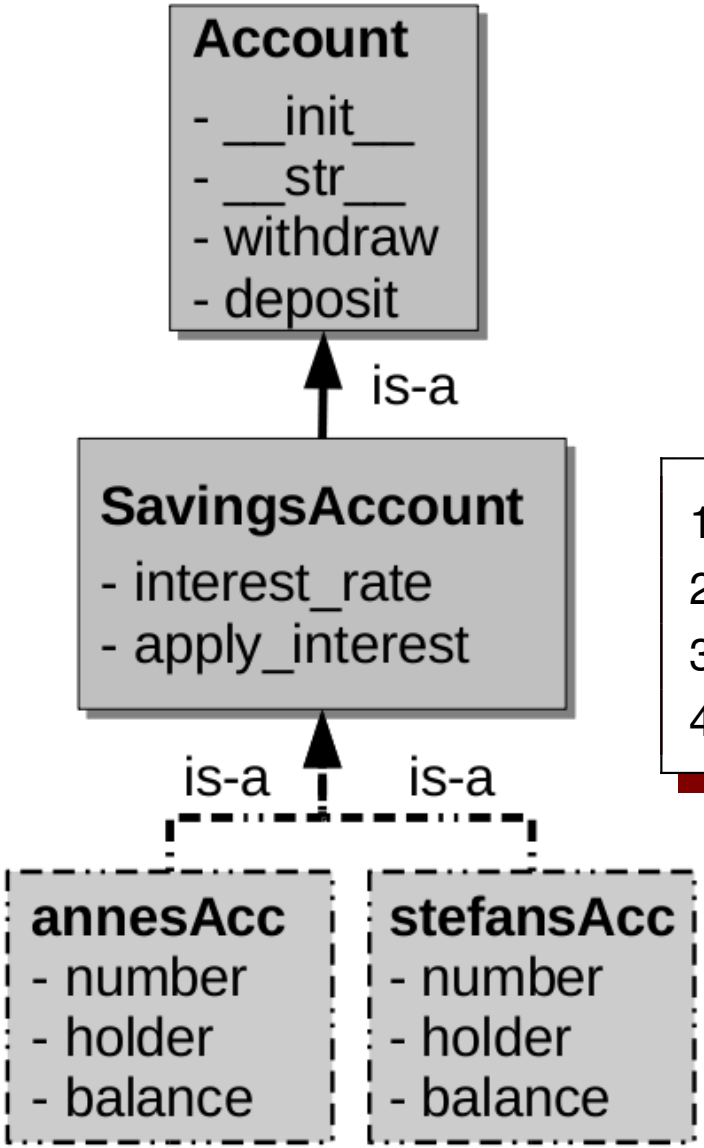
- **Savings Account**: We record the account number, holder and balance with each account. The balance has to be ≥ 0. We can apply an interest rate which is defined once for all savings accounts. Money can be deposited into the account. The account statement that can be printed includes the account number, 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. For instance, if the credit range determined for Anne is $500, her balance may not be less than - $500. Money can be deposited into the account. The account statement that can be printed includes the account number, holder and balance.

Class Design: On board

Commonalities/General Functionality: Base Classes

```
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
```

Special Cases: Derived Classes/Subclasses



- `SavingsAccount` is based on `Account`
- Methods of superclass are available in subclass (and objects created from subclass)

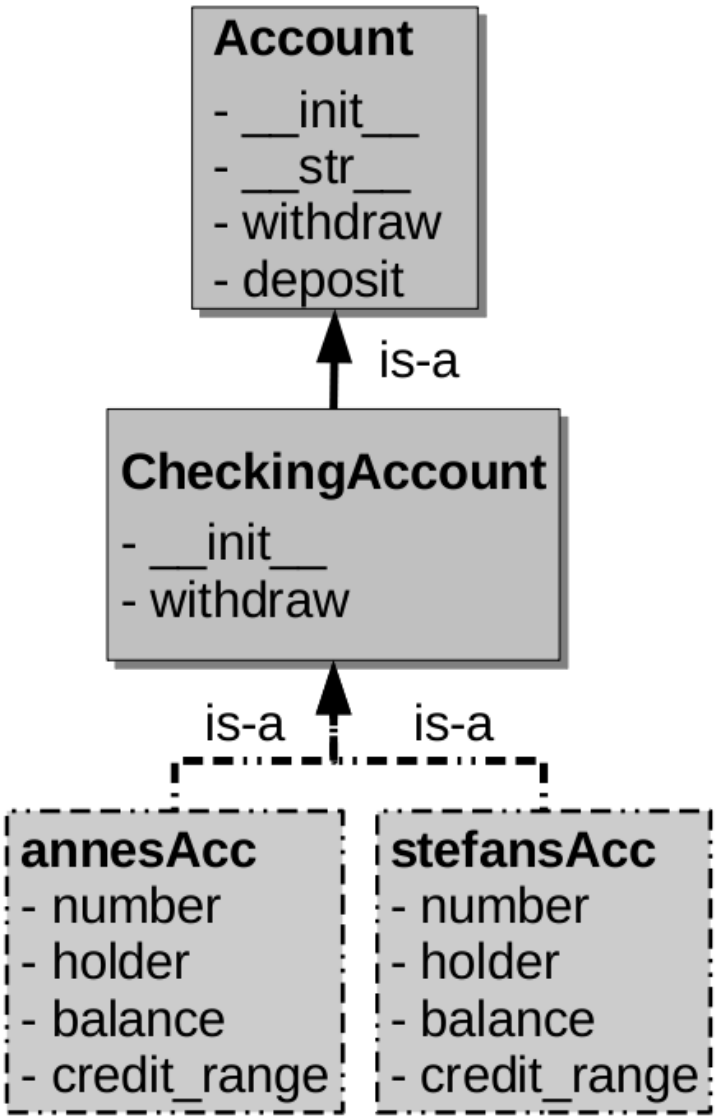
```
1 annesAcc = SavingsAccount(1, "Anne")
2 annesAcc.deposit(200)
3 annesAcc.apply_interest()
4 print(annesAcc)
```

Special Cases: Derived Classes/Subclasses

- `SavingsAccount` is based on `Account`
- `SavingsAccount` is derived from `Account`
- `SavingsAccount` extends `Account`
- Methods of superclass are available in subclass (and objects created from subclass)
- `SavingsAccount` provides some additional functionality

```
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)
```

Overriding Methods



- `Account` class also has `__init__` and `withdraw` methods
- `CheckingAccount` class overrides these

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

Which methods are called?

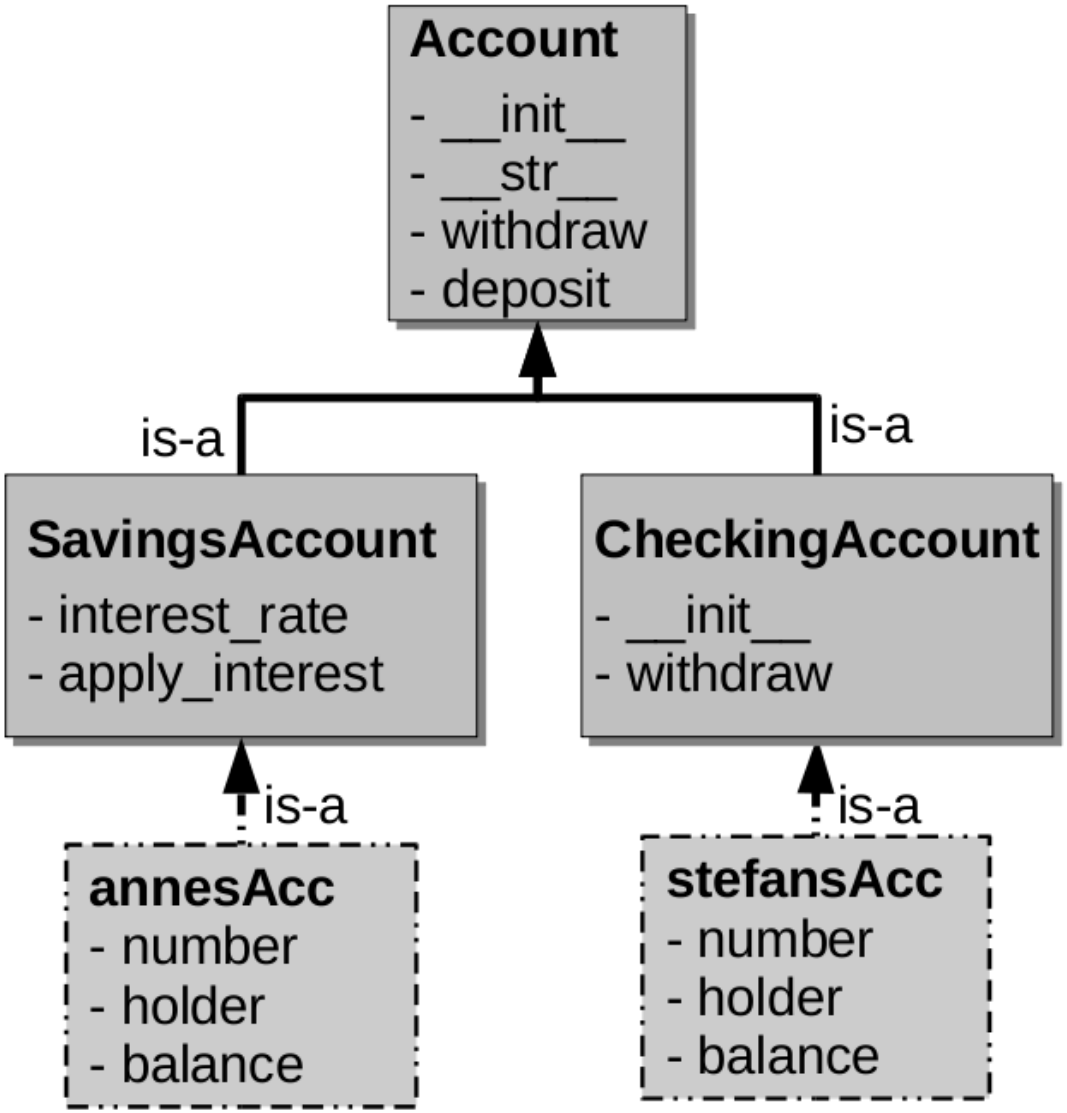
- `__init__`
- `deposit`
- `withdraw`
- `__str__`

Overriding Methods

- `Account` class also has `__init__` and `withdraw` methods
- `CheckingAccount` class overrides these

```
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
```

Class Hierarchy



Which methods are executed when calling them on `annesAcc` or `stefansAcc`?

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

Polymorphism

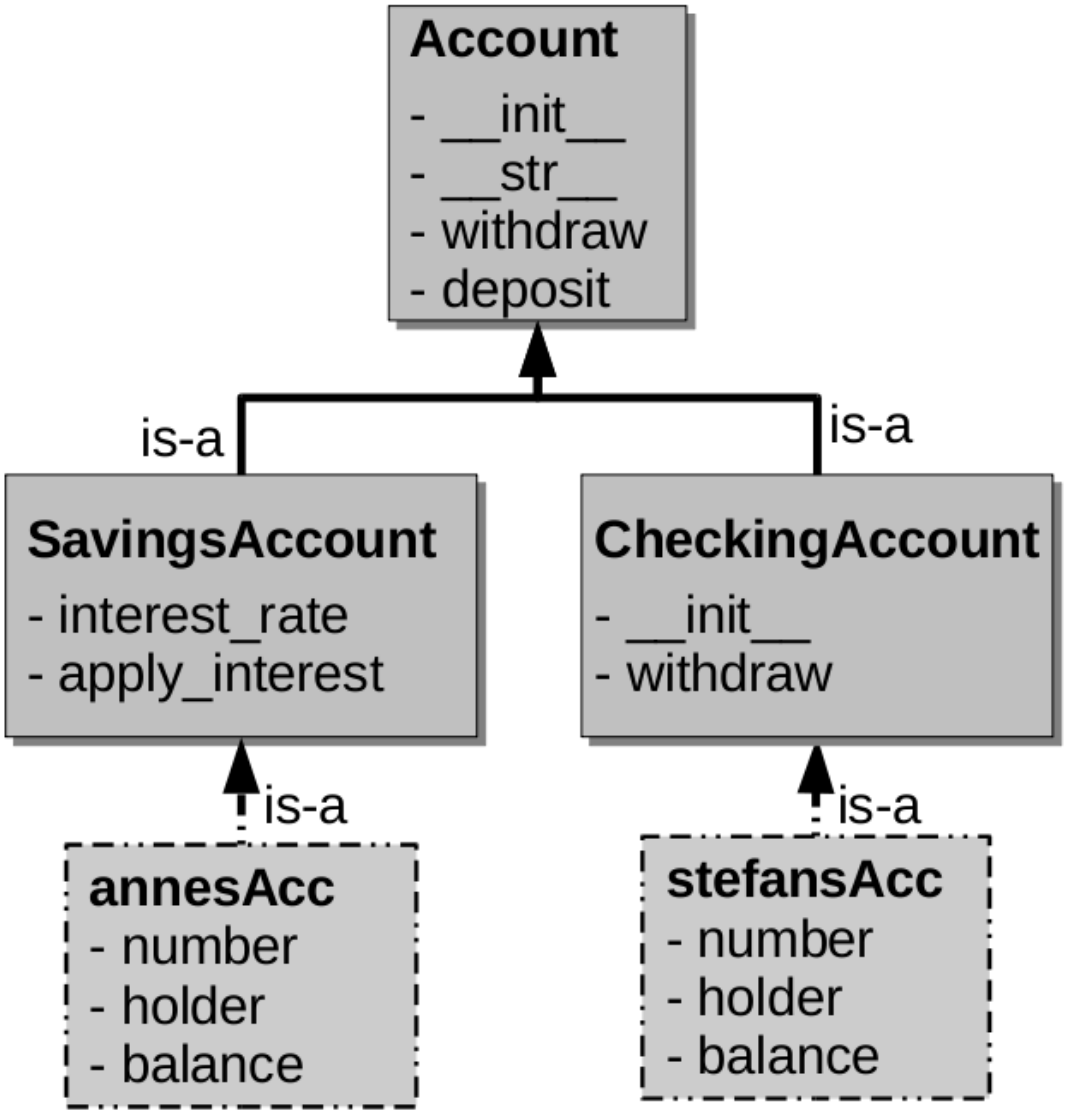
- 'having multiple forms' (Greek)
- When calling a method on objects, what happens depends on the class hierarchy from which the objects were created

Example

```
annesAcc.withdraw(400)
```

- We don't care what type of account `annesAcc` is
- Python follows the inheritance hierarchy & produces the desired behavior

Class Hierarchy Design



We could have defined `withdraw` twice - once in each subclass.

Why might it be useful to have it in the `Account` class?

Redundancy

- Data describing the same object exists twice (e.g. one person has two accounts, record holder information for each account separately)
  $\Rightarrow$ can result in inconsistencies
- The same code is written twice (or even more often)
  $\Rightarrow$ difficult to maintain

```
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
```

Minimizing Redundancy

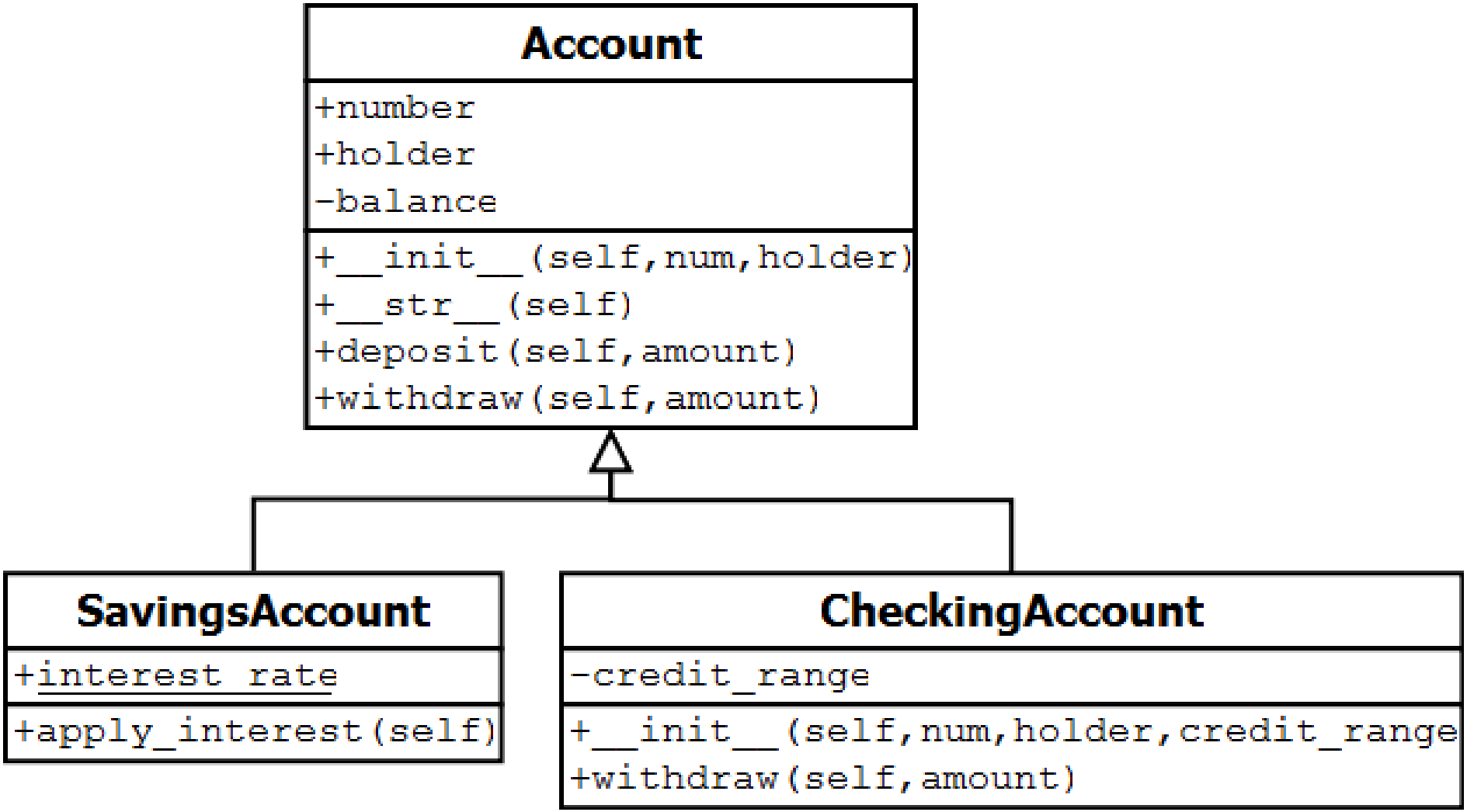
- Here: call the method of a superclass from the subclass
  $\Rightarrow$ extend the method of the superclass
- Method is called *on the class* $\Rightarrow$ need to pass reference to instance object to `self` of method in superclass explicitly here!

```
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
```

Minimizing Redundancy: Another example

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

UML Class Diagrams: Inheritance



Multiple Inheritance

- In some OOP language (e.g. Java) classes can only extend a single class.
- In Python, a class can inherit from more than one class
  $\Rightarrow$ Multiple Inheritance
- Special mechanisms to resolve which class's method is called
- This is a somewhat advanced feature.
- Recommendation: For now, let your classes have at most one superclass.

Everything in Python is an object

- We have used objects in Python right from the start of this course
- Lists and Dictionaries are objects
- Creation using special syntax (not by calling the class explicitly)
- Even strings and numbers are objects

```
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"
```

- Line 8 calls a hook method of the dictionary:
  `__setitem__(self, key, value)`

Everything in Python is an object

- We can even subclass the built-in classes
- Here: Overriding hooks

```
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
```

Understanding OOP is useful because...

- In real life, you will rarely program *from scratch*
- You will extend / customize other people's code
- Frameworks = collections of superclasses that implement common programming tasks
- You need to understand how the classes of the framework work together
- You write subclasses that specialize the behavior for your task
- Recipes for how to extend superclasses in an effective way:
  Design Patterns
  (*Design Patterns: Elements of Reusable Object-Oriented Software*
  by: Gamma, Helm, Johnson, Vlissides)

Remark on Terminology: Encapsulation

- Used for two aspects of OOP in the literature

1. **Data Encapsulation**: Data should be hidden (only accessed via instance methods)
2. **Encapsulation**: wrap up program logic behind interfaces (class names and the methods of the functions) such that each functionality is only defined once in a program

Congratulations: Now you're an object-oriented programmer

This week's exercise:

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.