Classes and Object Orientation

Object-oriented programming

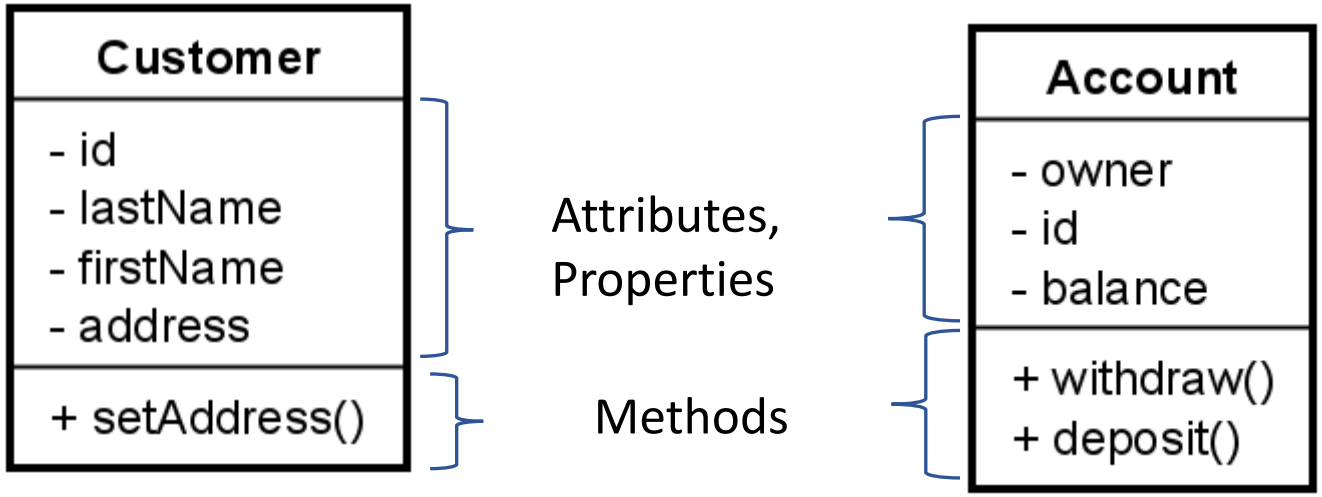
- Object-oriented programming summarizes data and its functions at one place.
- The data (information) of the objects is called **properties** or **attributes** of the object.
- The functions that can be applied to this data are called **methods**.

Object-oriented programming

- Objects are defined using classes.
- Classes are templates (blueprints) for objects.
- A class is a formal description of how an object is built
  - what attributes it has
  - what methods it has.
- Objects are created according to their template (the class).
- Instead of object, we also speak of instances of a class.

Example: Bank customer

- Customer and his account



Encapsulation

- Another major advantage of OOP is the encapsulation of the objects data.
- Access to the attributes of an object is done through access methods.
- Access methods can contain plausibility tests, data type conversions, or arbitrary calculation.
- You can also restrict the access (e.g. only read and no writing access).

The Customer class

The init method determines what attributes the class has and how it is backed by data.

```
# Customer of a bank
class Customer():
  #constructor method
  def __init__(self, id, lastName, firstName, address):
    self.__id = id
    self.__lastName = lastName
    self.__firstName = firstName
    self.__address = address
```

→ The other methods have to be added

The Account class

The init-method of the account class has no argument for the account balance. The account balance is zero at the beginning, so the parameter is unnecessary.

```
# account class
class Account():
  # constructor method to create new account objects
  def __init__(self, number, customer):
    self.__customer = customer
    self.__id = number
    self.__balance = 0
```

→ The other methods of the class have to be added.

Methods of Customer class

- In addition to the init method, the class needs access methods (get/set) for the attributes of the class.
- The customer id is immutable (no set-method).

```
def getId(self):
  return self.__id

def getLastname(self):
  return self.__lastName

def setLastname(self, name):
    self.__lastName = name

. . .

def getAddress(self):
  return self.__address

def setAddress(self, address):
  self.__address = address
```

Methods of Account class

The Account class has methods for the deposition and withdrawal of money.

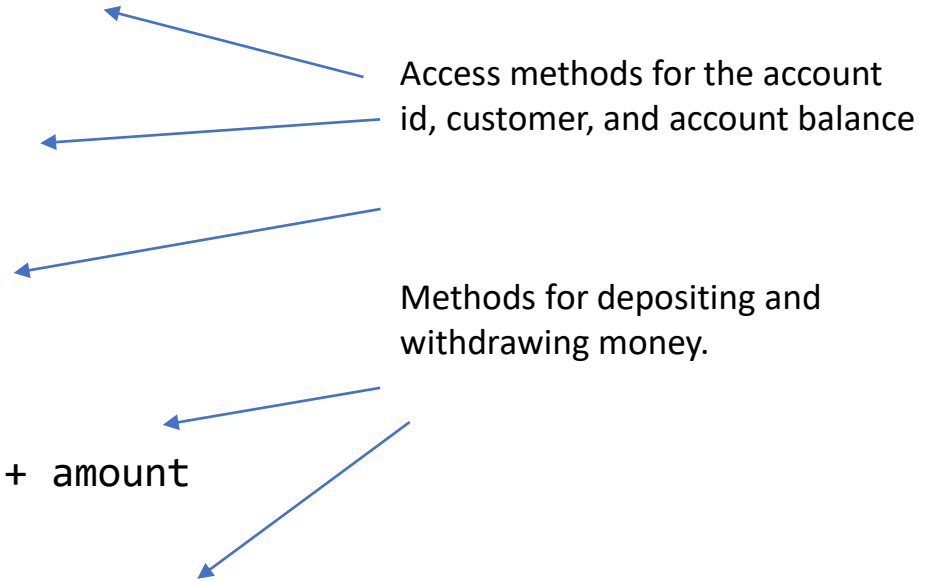
```
def getId(self):
   return self.__id

 def getCustomer(self):
   return self.__customer

 def getBalance(self):
   return self.__balance

 def deposit(self, amount):
   self.__balance = self.__balance + amount

 def withdraw(self, amount):
   self.__balance = self.__balance - amount
```



Usage of the Customer class

```
import Customer as c

# create new customer Peter
peter = c.Customer(17, "Peter", "Muster", "Bern")

# create new customer Julius
julius = c.Customer(102, "Julius", "Muster", "Bern")

# create new customer Julia
julia = c.Customer(103, "Julia", "Muster", "Bern")
```

Usage of the Account class

Create the Accounts and deposit and withdraw some money.

```
import Account as a

# create account for Peter Muster
k1 = a.Account(10, peter)

# create account for Julius Muster
k2 = a.Account(20, julius)

# create account for Julia Muster
k3 = a.Account(30, julia)
```

```
# deposit 5000 on Peters account
k1.deposit(5000)

# withdraw 50 from Julius account
k2.withdraw(50)
```

Magic Methods

Magic methods are internal methods that are indirectly called in special situations:

Create new objects

__init__ (. . .)

Converting objects to texts (to print them)

__str__ (. . .)

Explicit deletion of objects (clean up)

__del__ (. . .)

Usage in the Account class

Output of the account balances and their owners using a str method in the Account class:

```
def __str__(self):
   return str("Account No:" + str(self.__id) + " customer " +
              str(self.__customer) + ", balance " +
              str(self.__balance))
```

```
print(str(k1))
print(str(k2))
print(str(k3))
print(str(k4))
```

Ausgabe

```
Account No:10 customer Peter Muster Bern, balance 4950
Account No:20 customer Julius Muster Bern, balance 50
Account No:30 customer Julia Muster Bern, balance 150
Account No:40 customer Hans Muster Bern, balance 1800
```

Implementation of magic methods

```
class Bank():

  # create a new bank object
  def __init__(self):
    self.__customers = list()
    self.__accounts = dict()

  # delete all data
  def __del__(self):
    self.__customers = None
    self.__accounts = None

  # create string for printing
  def __str__(self):
    return str("Number of customers:" +
                  str(len(self.__customers)) +
               " Number of accounts:" +
                  str(len(self.__accounts)))
```

Bank class, further methods

```
# insert customer
def add_customer(self, customer):
    self.__customers.append(customer)
# insert account
def add_account(self, account,):
    self.__accounts[account.getCustomer()] = account
# find account with number nr
def get_account(self, customer):
    return self.__accounts.get(customer)
# get all accounts
def get_accounts(self):
    return self.__accounts.values()
```

Objects in the variable explorer

In the variable explorer we find all objects of class Account, Customer, Bank, …. with their values.

Name	Type	Size	Value
annasKonto	Account	1	Account object of Account module
customerListe	list	2	[Customer, Customer]
meineBank	Bank	1	Bank object of Bank module
peter	Customer	1	Customer object of Customer module
petersKonto	Account	1	Account object of Account module