UPPSALA
UNIVERSITET

Object-oriented analysis and programming

Advanced programming

March 9, 2021

Artur Kulmizev

UPPSALA
UNIVERSITET

Object orientation

Classes

Instance and class variables

Instance and class methods

Inheritance and polymorphism

Encapsulation

Abstract classes

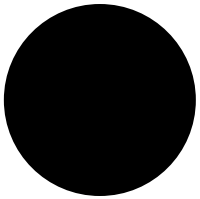
UML diagrams

UPPSALA
UNIVERSITET

Classes and Objects

Classes describes objects

```
class Circle{
      ....
}
```



```
class Square{
      ....
}
```

UPPSALA
UNIVERSITET

example - Person

```
class Person:

    # constructor
    def __init__(self, name, age):
      self._name = name
      self._age = age
```

Names with underscores denote variables that are meant to be treated as private, i.e. just to be used within the class

Double underscores can be used for the same purpose (name mangling as a result)

UPPSALA
UNIVERSITET

Instance variables

Variable with a unique value for each instance of the class

e.g. in the Person class:

```
_name
_age
```

UPPSALA
UNIVERSITET

Encapsulation

Protect variables so that other classes cannot use them directly

Not strictly implemented in Python, rely on conventions!

Start the name with an underscore

Why is it useful?

–Modularity: Hide implementational details

–Security: Allow control of values

–Flexibility: Allow changing variables types et.c.

UPPSALA
UNIVERSITET

Encapsulation – Java style

Use strictly private variables

–Or protected: visible in a sub classes also

Use public methods to get and set values:

–Get methods – `getAge(self)`

–Set methods – `setAge(self, age)`

Use public methods for other manipulation of the objects

UPPSALA
UNIVERSITET

Encapsulation – Python style

```
class Person:
      ...
    @property
    def age(self):
        return self._age
    @age.setter
    def age(self, age):
        if age <0 or age > 150:
            #Do something!
        self._age = age
```

UPPSALA
UNIVERSITET

Class variables / Static variables

Variable for the full class, has only one value for all Instances, but can be accessed from each instance

All instance of a class share a class variable
Also called **static variables**

example:

```
adultAge = 18
```

UPPSALA
UNIVERSITET

The Person class

```
class Person:

    adultAge = 18

    # constructor
    def __init__(self, name, age):
      self._name = name
      self._age = age

    ...
```

UPPSALA
UNIVERSITET

Class methods / Static methods

A method that cannot use instance variables.

Marked with decorators:

```
@classmethod

or

@staticmethod
```

UPPSALA
UNIVERSITET

The Person class

```
class Person:

    adultAge = 18

    # constructor
    def __init__(self, name, age):
        self._name = name
        self._age = age

    @classmethod
    def getAdultAge(cls):
        return cls.adultAge

    def isAdult(self):
        return self._age >= Person.adultAge
```

UPPSALA
UNIVERSITET

Class methods

```
@classmethod
def getAdultAge(cls):
    return cls.adultAge
```

Class methods are useful when you need to access class variables

`cls` argument is similar to self, but relates to the full class

```
@classmethod
def fromBirthYear(cls, name, byear):
    return cls(name, date.today().year-byear)
```

Useful also for “factory” methods, where we create a new instance of the class

UPPSALA
UNIVERSITET

Static methods

```
@staticmethod
def returnClassification():
    return {“genus”: “homo”;
            ”species”: “sapien”}
```

Static methods usually do not take any arguments

Mainly useful for utility functions,
(they are added to the class namespace)

It is possible to access static variables,
but classmethod is recommended for that

```
@staticmethod
def getAdultAge():
    return Person.adultAge
```

UPPSALA
UNIVERSITET

Calling Class/Static methods

```
@staticmethod
def returnClassification():
    return {“genus”: “homo”;
            ”species”: “sapien”}

@classmethod
def getAdultAge(cls):
    return cls.adultAge

scott = Person(“Scott”, 45)

scott.getAdultAge()              #both options
Person.getAdultAge()             #possible

Person.returnClassification()  #only option
```

UPPSALA
UNIVERSITET

UML

Unified Modeling Language

A tool to express and model ideas

–A graphical description of different aspects of object oriented systems

Class diagrams

–Describe classes and relations between classes

UPPSALA
UNIVERSITET

UML diagram types

Class diagram

Use case diagram

Sequence diagram

Communication diagram

UPPSALA
UNIVERSITET

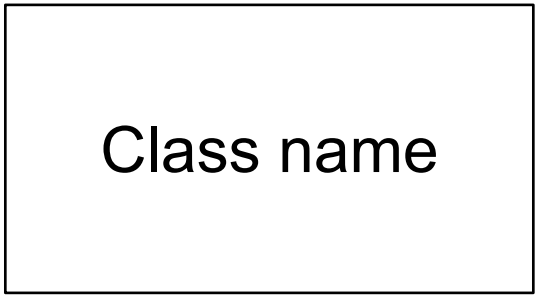
UML – class description

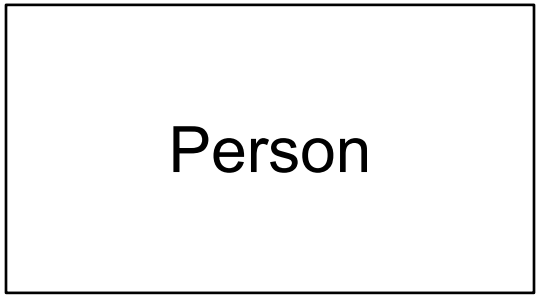
Class name
variables
methods

Person

UPPSALA
UNIVERSITET

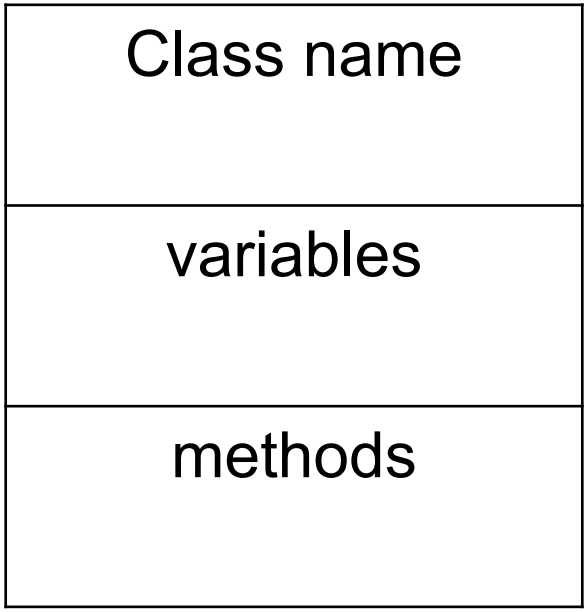
UML – *simplified* class description

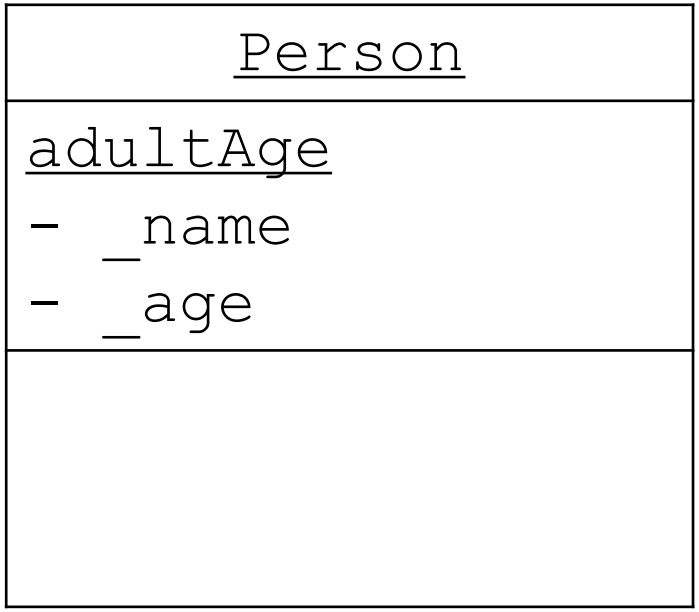




UPPSALA
UNIVERSITET

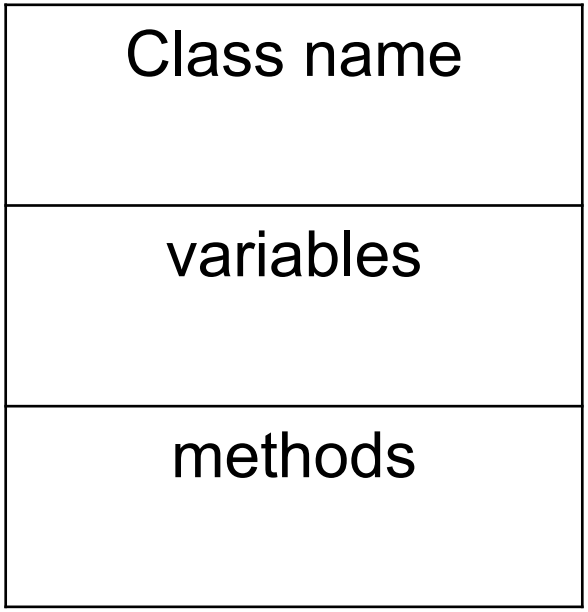
UML – class description

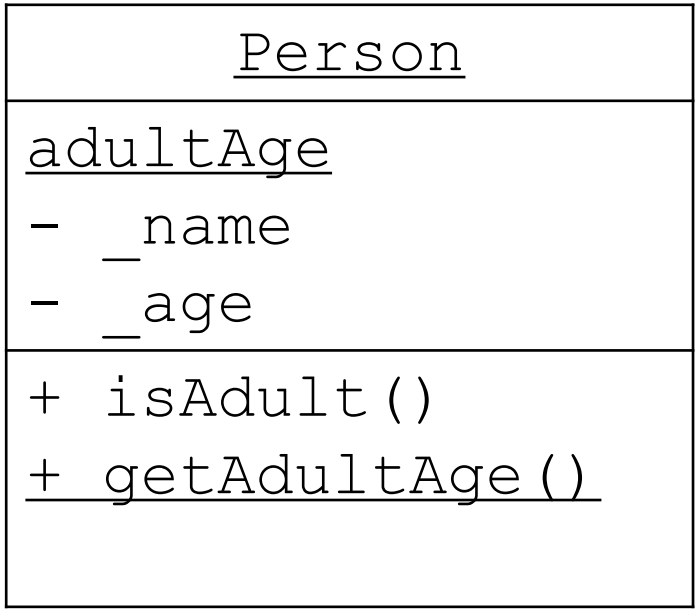




UPPSALA
UNIVERSITET

UML – class description





UPPSALA
UNIVERSITET

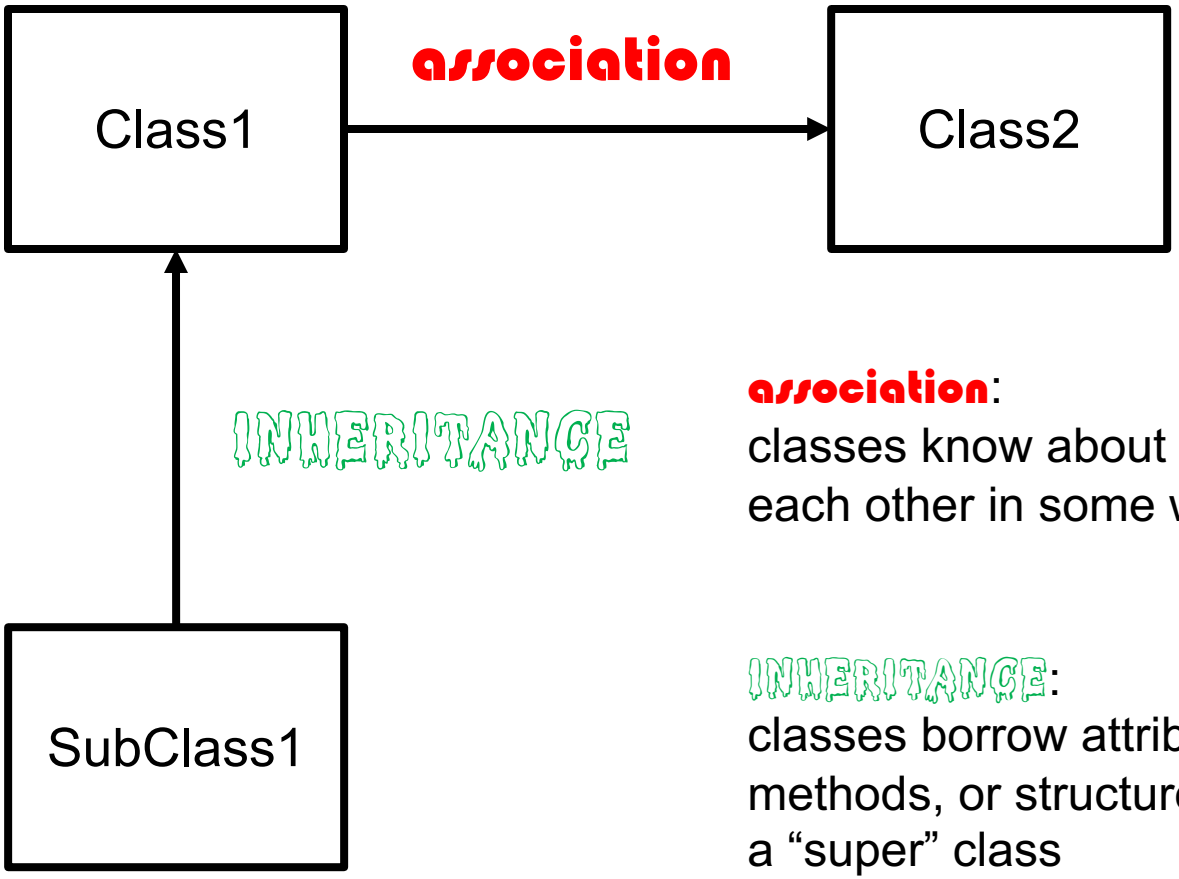
UML – class description

Person
adultAge - _name - _age
+ isAdult() + getAdultAge()

Instance variable
Instance method
Class variable
Class method
- private
+public

UPPSALA UNIVERSITET

UML – relations between classes



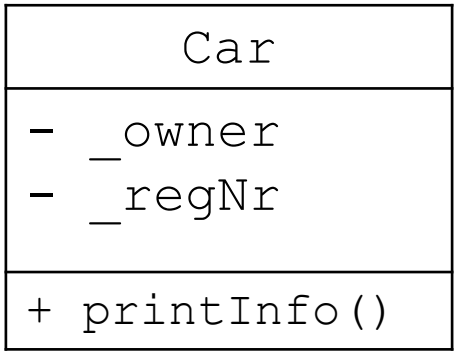
association:
classes know about each other in some way

INHERITANCE:
classes borrow attributes, methods, or structure from a “super” class

UPPSALA
UNIVERSITET

The class Car

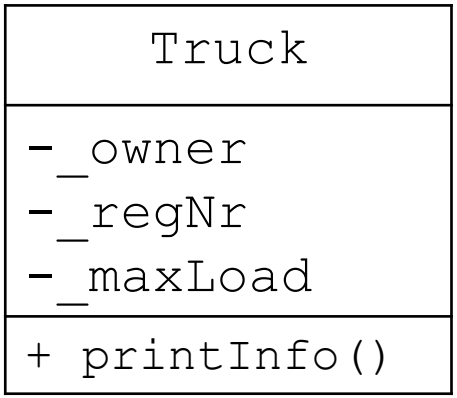
```
class Car:

    def __init__(self, owner, reg):
      self._owner = owner
      self._regNr = reg

    def printInfo(self):
      print ("Car: ", self._regNr)
```

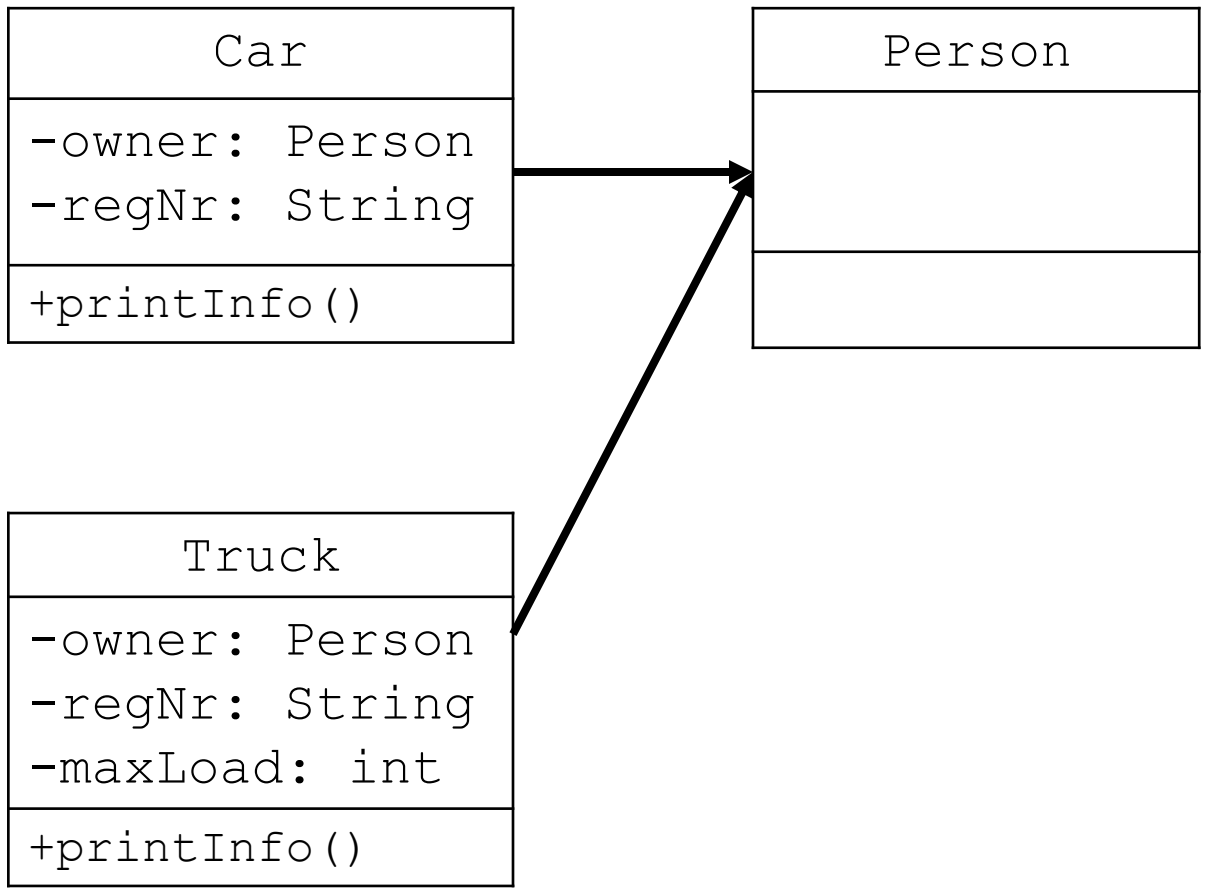


UPPSALA
UNIVERSITET

The class Truck

```
class Truck:

    def __init__(self, owner, reg, load):
      self._owner = owner
      self._regNr = reg
      self._maxLoad = load
    }

    def printInfo(self) {
      print ("Truck: ", + self._regNr, \
      ", max load: ", self._maxLoad)
    }
}
```




Car
-owner: Person
-regNr: String
+printInfo()
Person
Truck
-owner: Person
-regNr: String
-maxLoad: int
+printInfo()

UPPSALA
UNIVERSITET

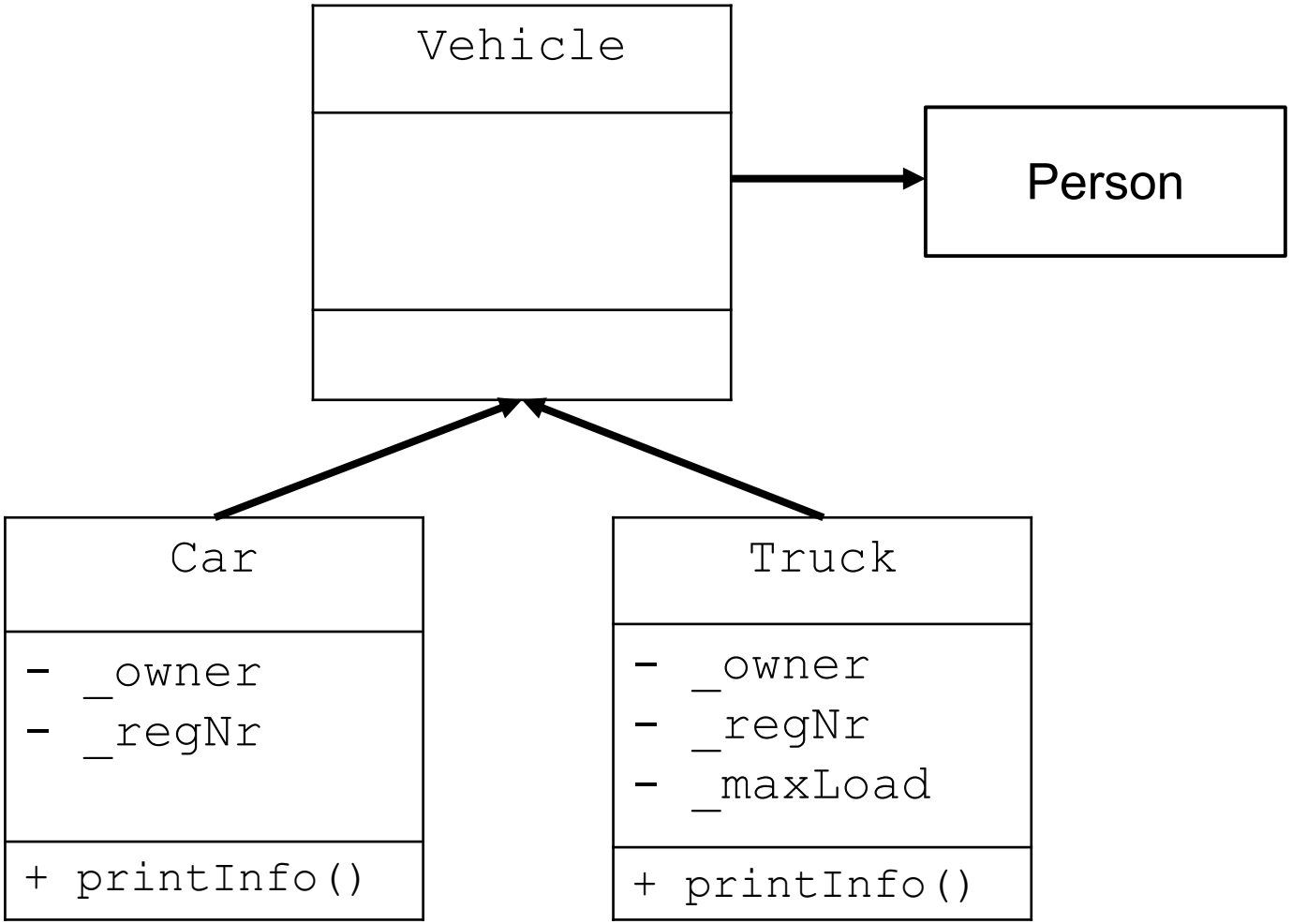
```
scott = Person(“Scott”, 45)
scottsCar = Car(scott, “BBC123”)
scottsTruck = Truck(scott, “TBC123”, 100)

myCar.printInfo())
myTruck.printInfo())
```

Printout:

```
Car: BBC123
Truck: TBC123, max load: 100
```

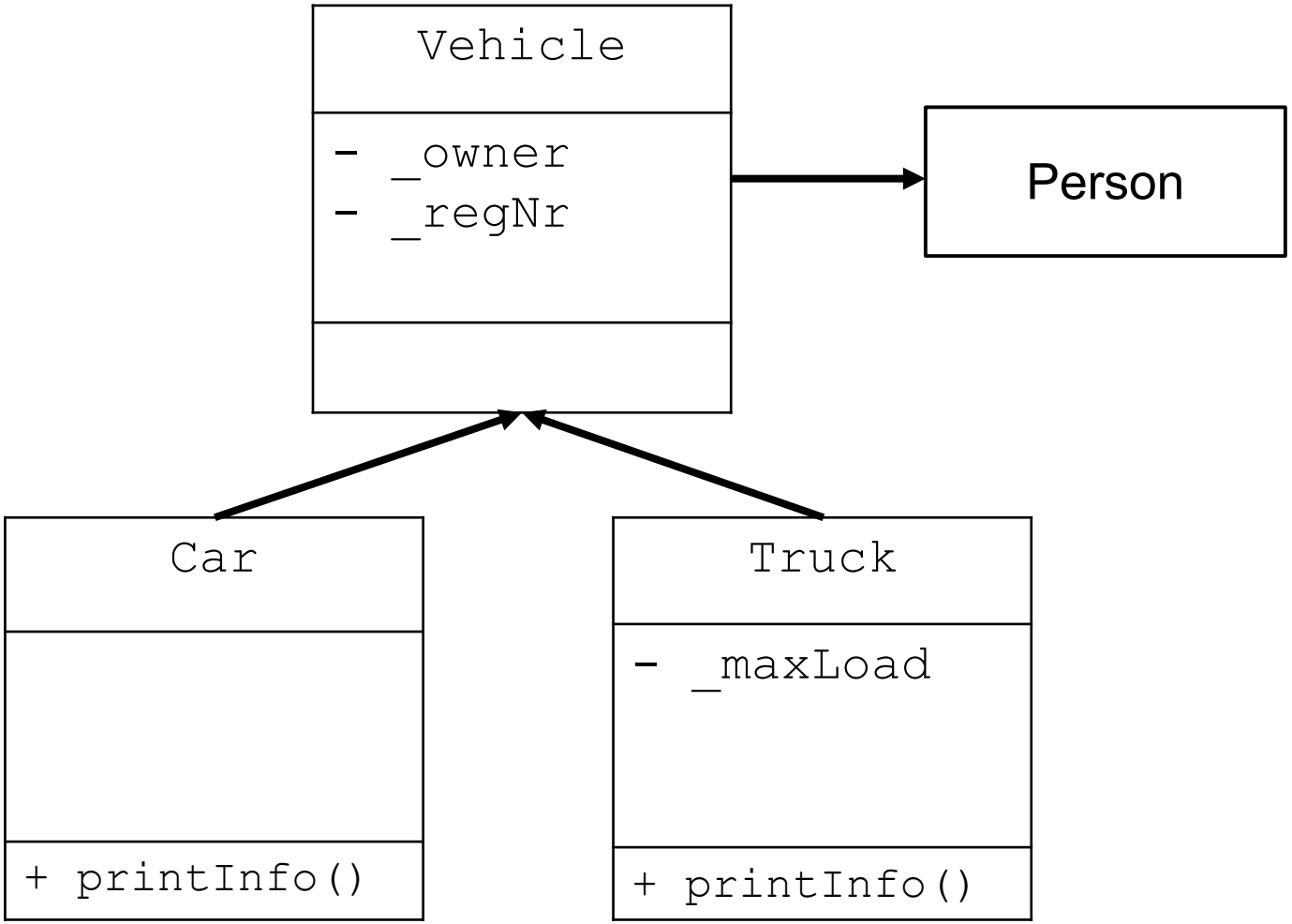
UPPSALA
UNIVERSITET

INHERITANCE – both classes are vehicles



UPPSALA
UNIVERSITET

INHERITANCE – both classes are vehicles



UPPSALA UNIVERSITET

New code with INHERITANCE

```
class Vehicle:
    def __init__(self, o, r):
        self._owner = o
        self._regNr = r


class Car(Vehicle):
    def __init__(self, o, r):
        super().__init__(o, r)


class Truck(Vehicle):
    def __init__(self, o, r, l):
        super().__init__(o, r)
        self._maxLoad = l
```

super keyword used to access parent class!

UPPSALA
UNIVERSITET

Visibility

Python does not have strict typing

Typically subclasses are also expected to be allowed to use private (`_name`) variables

In other languages:

–Private – only in class

–Protected – also in subclasses

–Public – everywhere

UPPSALA
UNIVERSITET

Add super class method – solution 1

```
class Vehicle:

    def __init__(self, owner, reg):
      self._owner = owner
      self._regNr = reg

    def printInfo(self):
      print(Reg nr: ", self._regNr)
```

`printInfo()` in `Vehicle` will likely never be used

UPPSALA
UNIVERSITET

Add super class method – solution 2

```
class Vehicle:

    def __init__(self, owner, reg):
        self._owner = owner
        self._regNr = reg

    @abstractmethod
    def printInfo(self):
        pass
```

If we declare a method as abstract, the class becomes an abstract class, and we cannot create any instances of it

Any subclasses are required to implement all abstract methods!

UPPSALA UNIVERSITET

Add super class method – solution 3

```
class Vehicle:

    ...

    @abstractmethod
    def printInfo(self):
        print("Reg nr: ", self._regNr)

class Truck(Vehicle):
    def printInfo(self):
        super().printInfo()
        print(" max load: ", self._maxLoad)
```

An abstract class can contain an implementation that can be used in subclasses, using the keyword SUPER

UPPSALA
UNIVERSITET

POLYMORPHISM

Different methods are called depending on which subclass is called, even if we do not know what the subclass actually is

Example:

–If we have a list with both trucks and cars, and call `printInfo()` on each item, we will automatically use the correct variant, depending on if the object is a Truck or a Car

UPPSALA
UNIVERSITET

Duck typing

In Python, this is not restricted to inheritance hierarchies

As long as classes we want to call have the same interface, we can call the same method on them

–Duck typing

Duck typing is not restricted to inheritance hierarchies

UPPSALA
UNIVERSITET

Inheritance - terms

Super class – The class you inherit from, e.g. Vehicle

Sub class – The class that inherits from a super class, e.g Car and Truck

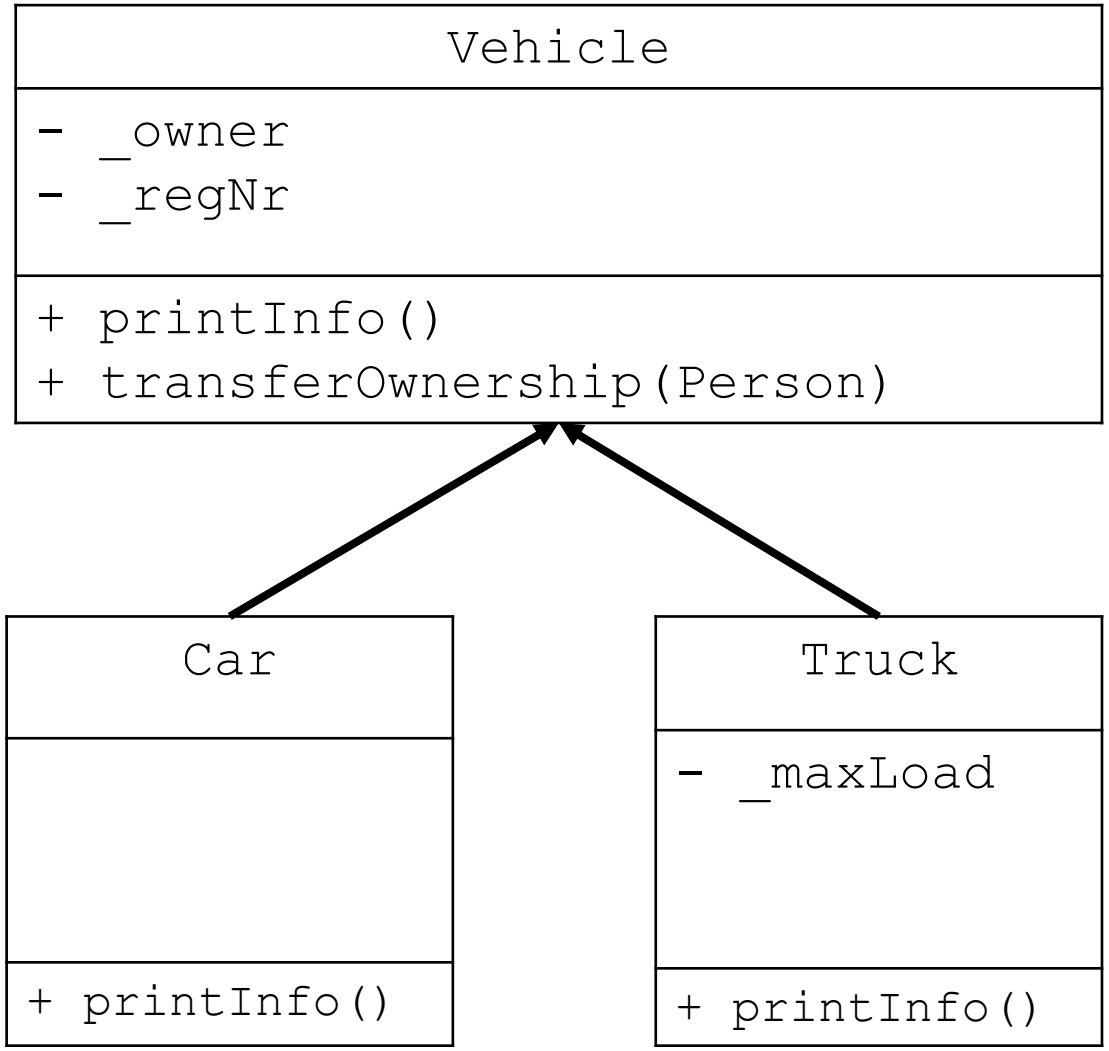
Simple inheritance – When a class inherits only from one super class (the normal case)

Multiple inheritance – When the class inherits from more than one super class (Caution needed, can be error prone, duck typing often better alternative)

SUPER - keyword used to access the super class

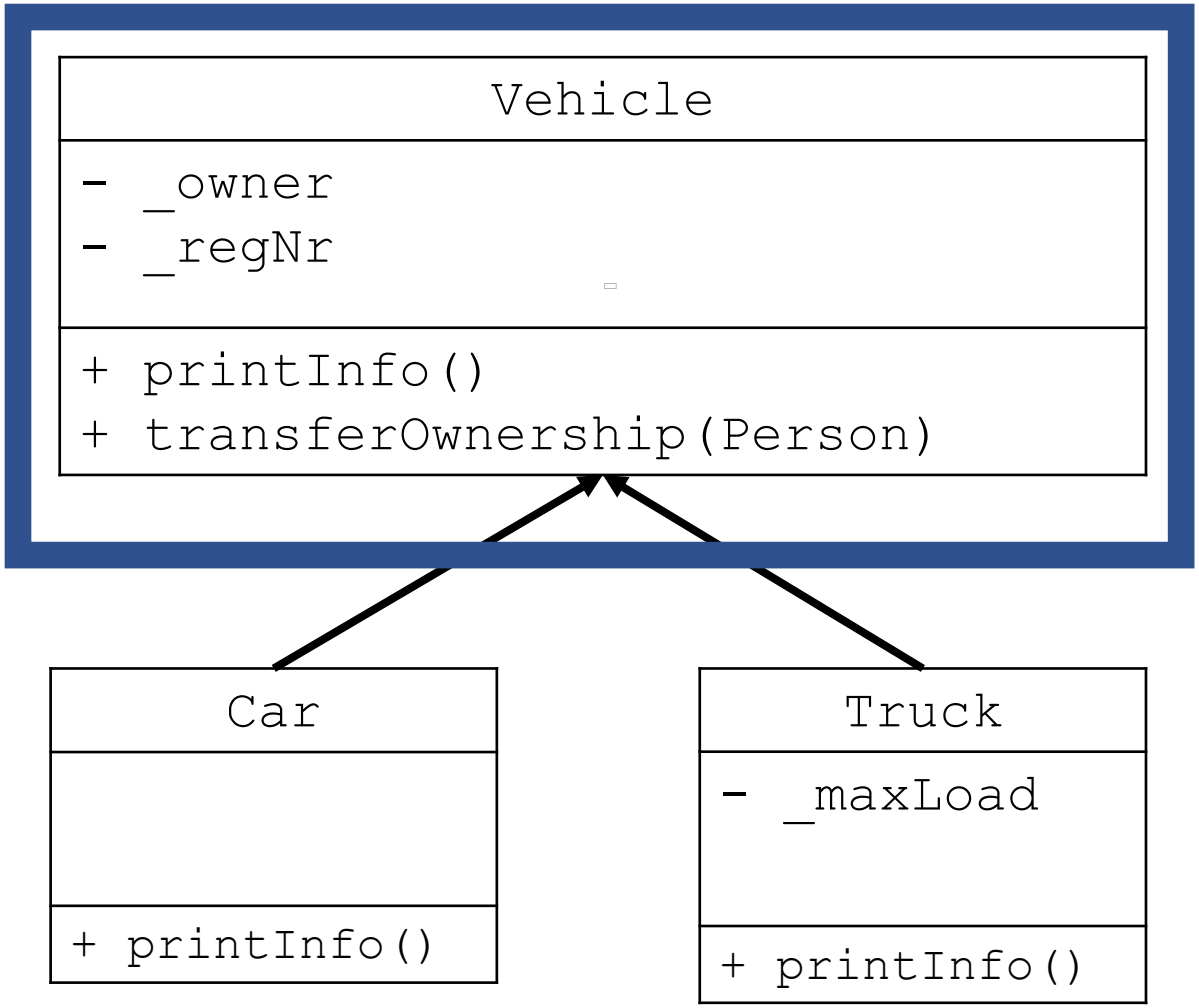
UPPSALA
UNIVERSITET

INHERITANCE – what sees what?



UPPSALA UNIVERSITET

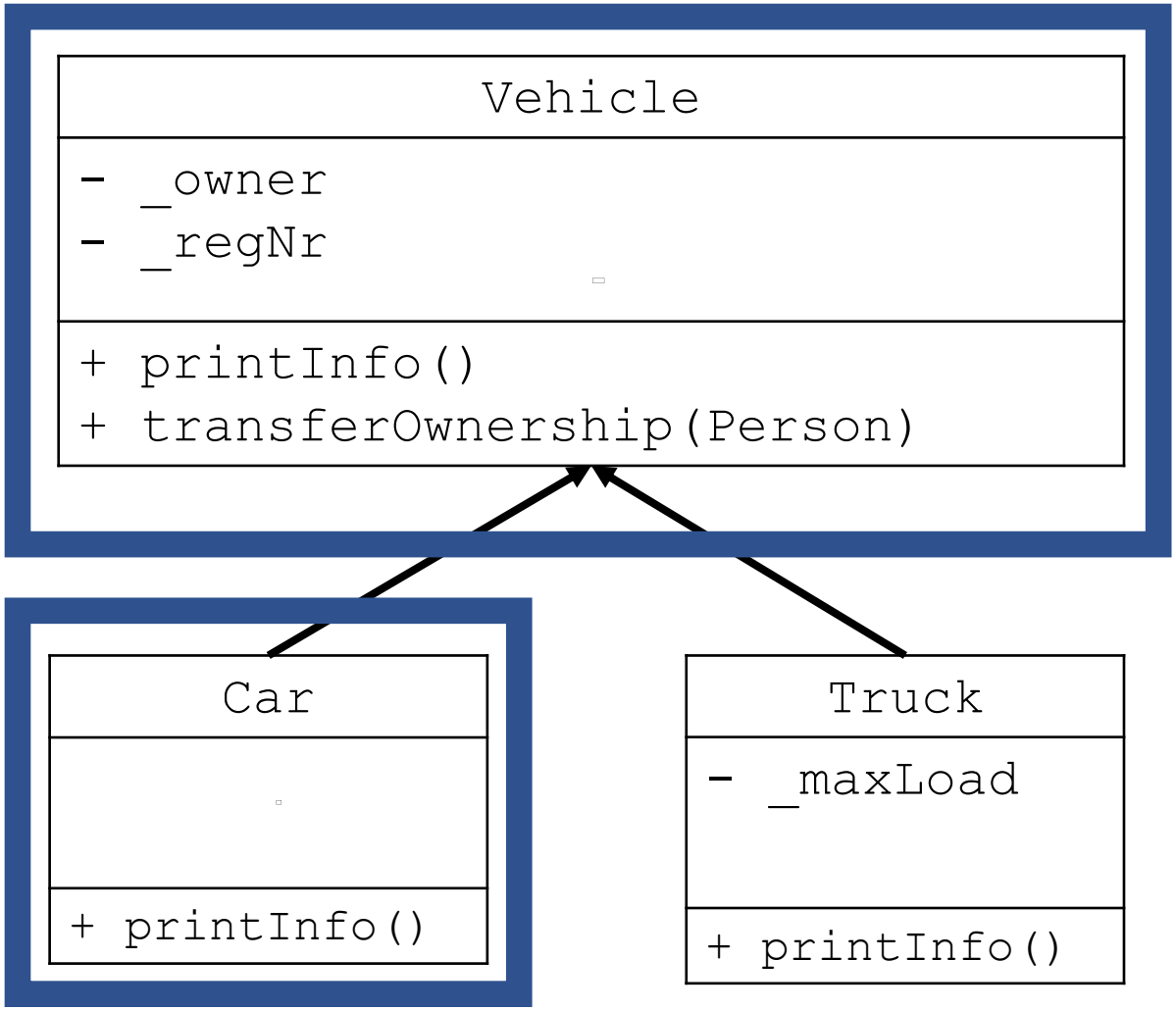
INHERITANCE – what sees what?



The `Vehicle` Class only sees itself!

UPPSALA UNIVERSITET

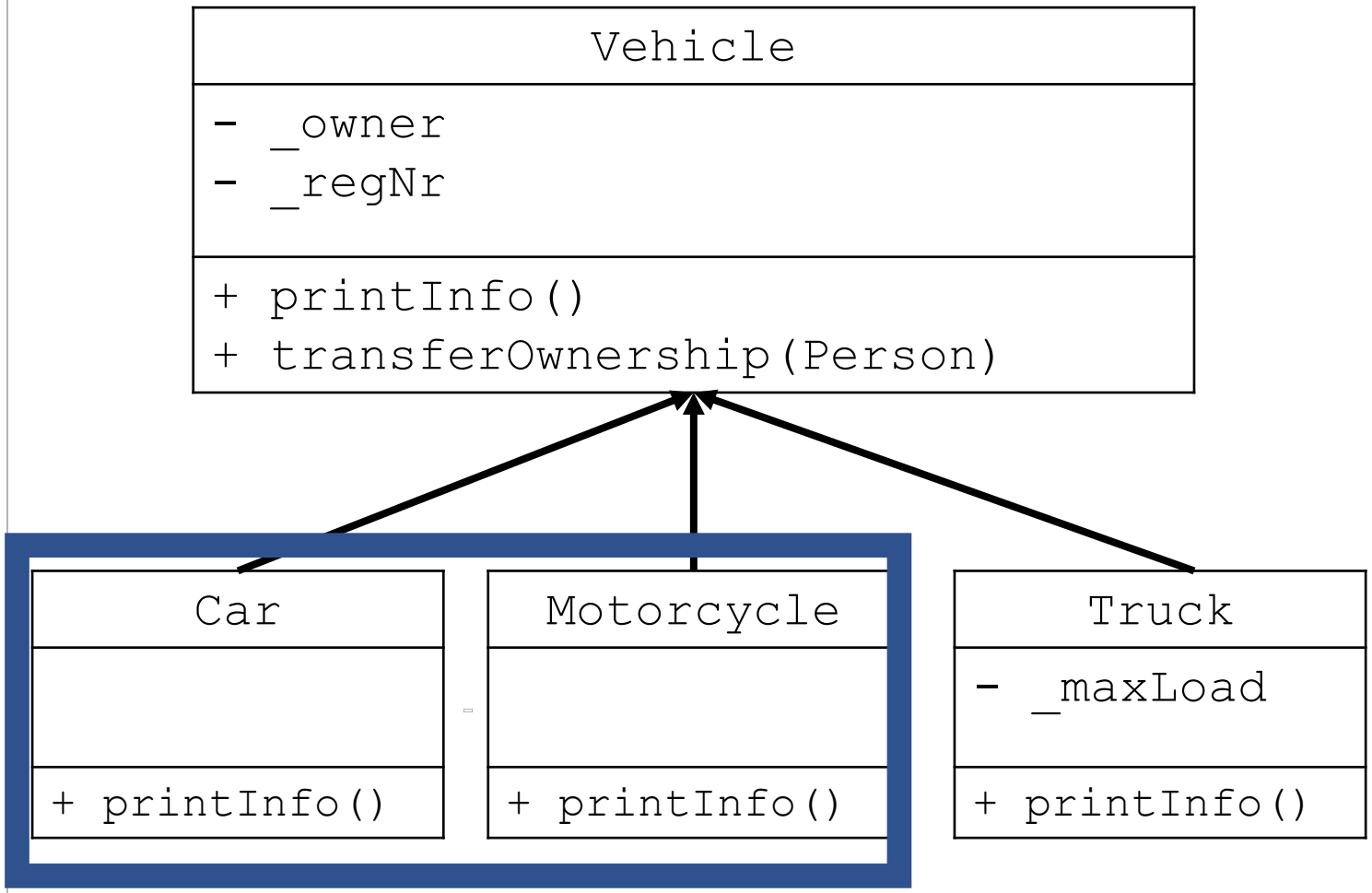
INHERITANCE – what sees what?



The `Car` Class sees itself and `Vehicle`!

UPPSALA UNIVERSITET

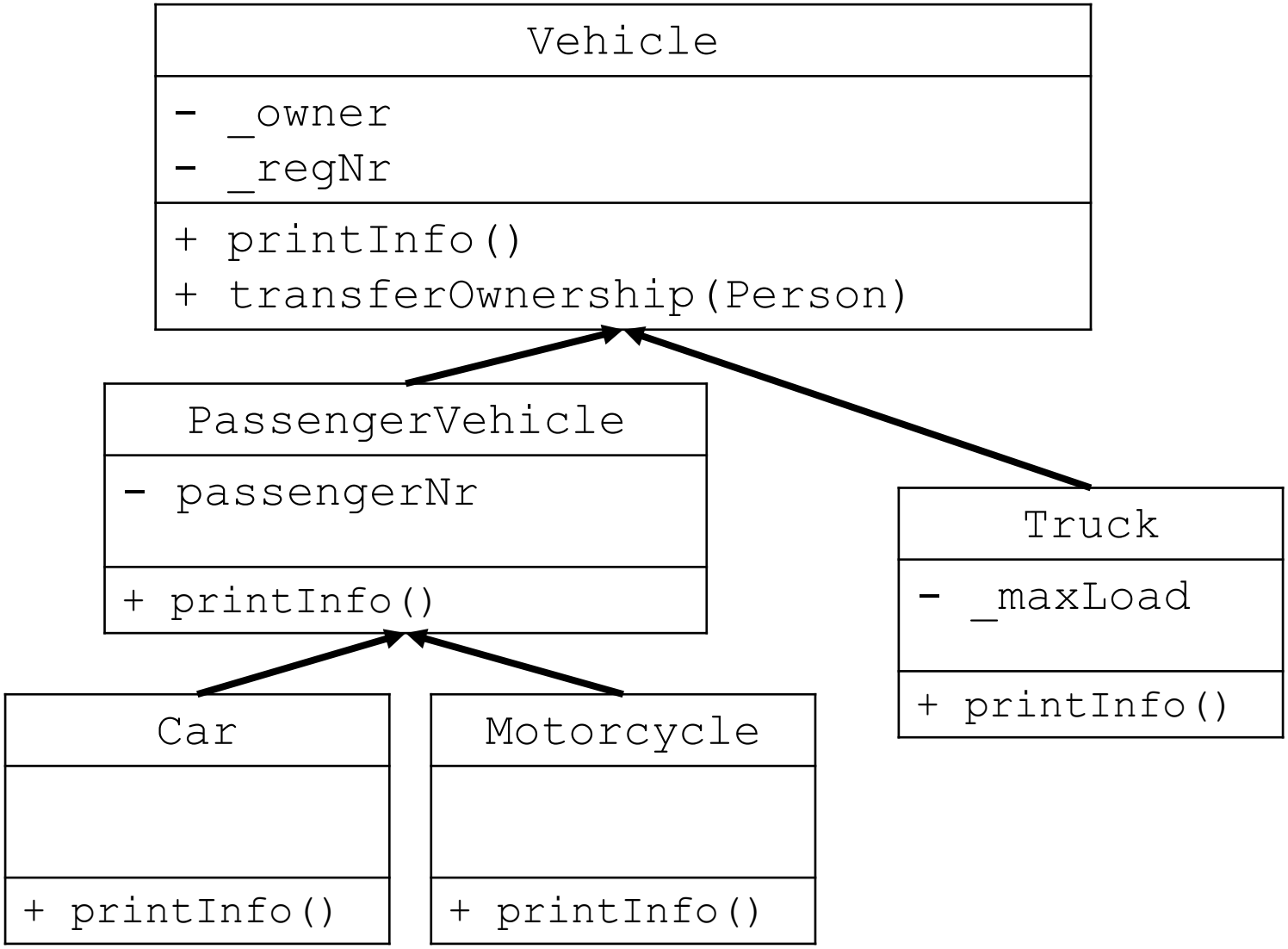
INHERITANCE – can we have deeper hierarchies?



The `Car` and `Motorcycle` classes can have passengers...

UPPSALA UNIVERSITET

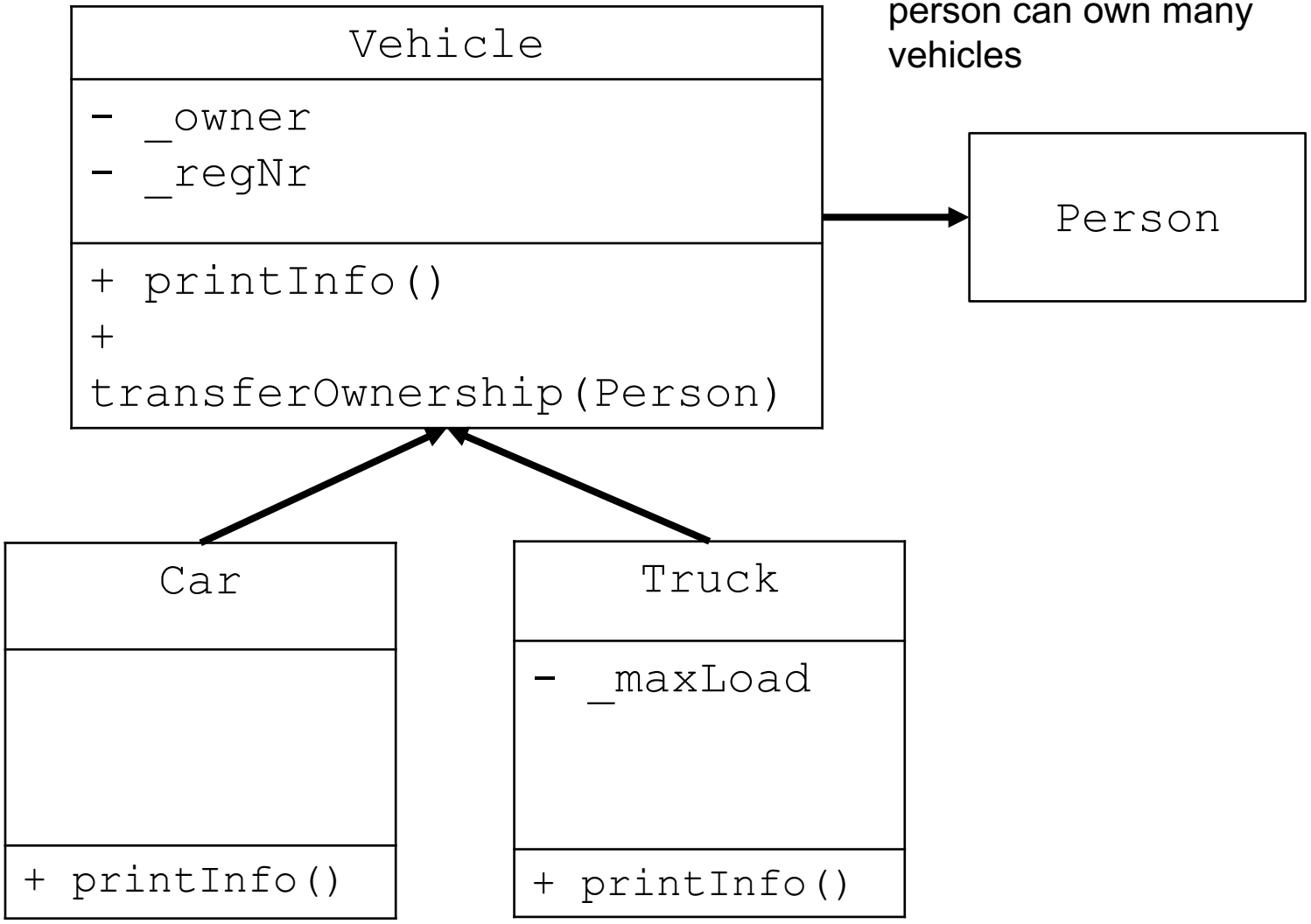
INHERITANCE – can we have deeper hierarchies?



UPPSALA UNIVERSITET

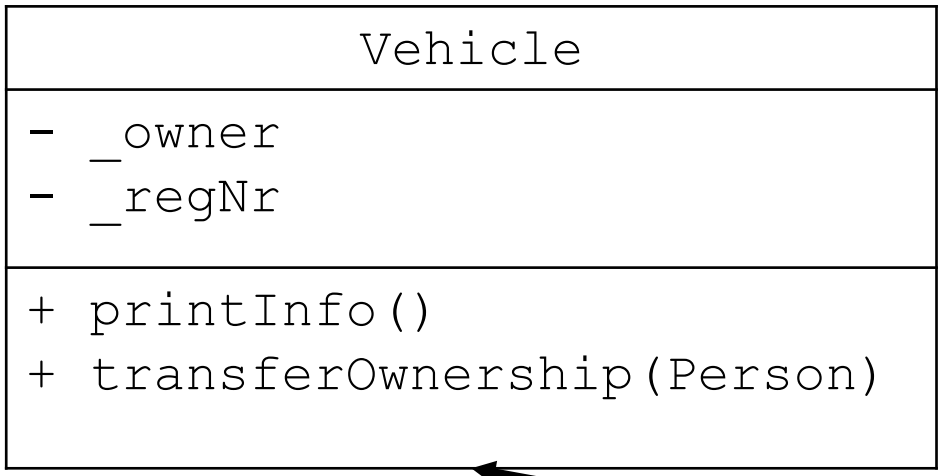
multiplicity

A vehicle can only be owned by a single person, whereas a person can own many vehicles



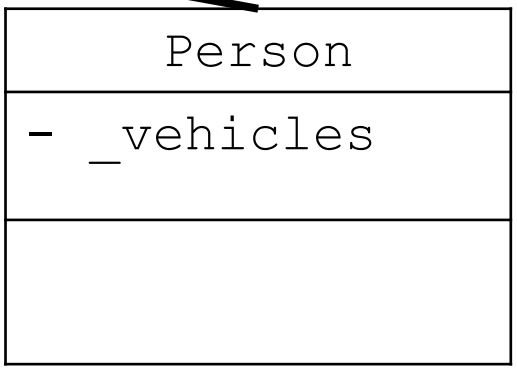
UPPSALA UNIVERSITET

multiplicity



The `Person` class now knows about which vehicles a person owns, rather than the vehicles knowing who owns them

`_vehicles` could be implemented as a list of `Vehicle` objects



UPPSALA
UNIVERSITET

OO[ADP]

OOA (object oriented analysis)

–Analyze the problem

–Create use cases – what does the program need to do?

–Decide which classes are needed and how they should cooperate

OOD (object oriented design)

–Make the design more specific and detailed

–Decide on methods and variables for each class

OOP (object oriented programming)

–Implement the design in some object oriented programming language

UPPSALA
UNIVERSITET

OOA/OOD – possible workflow

Find object candidates

Make a first cleanup among objects

Classify the objects

Group objects and describe relations between them

Define use cases and validate the system

Draw class diagrams

Design classes in detail

UPPSALA
UNIVERSITET

Find object candidates

Noun-verb method

–Nouns – possible classes

–Verbs – possible methods

Checklist

–Physical objects (car, house)

–Locations (room, garden)

–Concepts (bank account, transaction)

–Roles (customer, player)

–Events (landing, break)

–…

UPPSALA
UNIVERSITET

First cleanup

Are some objects really the same, but with different names?

–e.g. Agent, Player

Are some objects obviously implausible, i.e. should maybe be variables instead

–e.g. Name, can probably be an instance variable instead of a class

UPPSALA UNIVERSITET

Classify the objects

Work for instance with CRC-cards

–C – class, the name of the class

–R – responsibility of the class

–C – collaborators, classes that the class collaborate with

Create CRC cards for your class candidates

CRC-cards can be moved around, grouped, updated, thrown away as you continue working

Name

Responsibilities Collaborators

UPPSALA
UNIVERSITET

Use cases and validation

Use cases

–Specifications of what the program should do

Validation

–Can the classes be used to perform the use cases?

–Have we found the right classes?

–Have we identified all classes needed?

–Which classes cooperate, and how?

UPPSALA
UNIVERSITET

Design classes in detail

Which methods are needed?

–Especially public methods – the interface of the class

What variables are needed?

–Which data structures would be a good fit?

UPPSALA
UNIVERSITET

Assignment 4

Text-based game based on OOD

Should be fully object-oriented

–i.e. all code should be in methods in classes

–Except a very small main program, like for instance:

```
# get filename from command line args
reader = HouseReader()
myHouse = reader.initialize(filename)
myHouse.play()

# or
myHouse = House(filename)
myHouse.play()
```