Programming in Python

OOP principles, metamethods, relations between classes
lecture 7

Department of Cybernetics and Artificial Intelligence
Technical University of Košice
Ing. Ján Magyar, PhD.

OOP principles

1. abstraction
2. encapsulation
3. inheritance
4. polymorphism

Abstraction

- using abstraction we hide the implementation details regarding the functionality
- each class should provide only an API – a set of methods for working with class fields
- other objects and programmers should not care about how the given method works, only what it does
- for a high level of abstraction it is necessary to provide documentation, especially if the method has side effects

Abstraction in Python

1. when using classes, we don’t care how they implement the functionality
   - we rely on the documentation and the author of the code
2. since the class defines an abstract data type, we don’t care about the inner data representation
   - e.g. a hash table can be represented as a dictionary or a list of lists
   - we consider only the outer context of the class and not internal details

Abstraction - example

For representing a hash table, we can use a dictionary. If the hash function returns integer values, e.g. remainder after dividing by 6, possible keys are 0, 1, 2, 3, 4, 5.

We can change the representation into a list of lists:

- using the first index we select the list under the given hash value
- values will be stored under the second index in the selected list

Encapsulation

- while abstraction hides implementation details, the goal of encapsulation is to hide the internal state of an object
- the internal state is defined as the set of values in class fields
- encapsulation defines how we can work with objects of the given class

Encapsulation in Java/C# vs. in Python

- C-based object-oriented languages support encapsulation implicitly
- for each attribute we can define visibility using the keywords: `public`, `private`, `protected`, `package-private`
- Python does not have these keywords, it doesn't even have packages, the programmer must implement encapsulation explicitly
- best practice: within the class access fields directly, outside the class using helper methods

Private attributes in Python

- we can define private attributes within a class using __

```
class Product:
    def __init__(self):
        self.__price = 1000

t = Product()
t.__price = 500
```

Object state

- the object state is the set of values of each field in the given object
- two main types of fields:
  - object fields – unique for each object
  - class fields – unique for the entire class
- the object state changes during runtime

Class variables

- variables like `self.name` are specific for each instance
- Python enables the use of variables, which are shared by all instances of the given class
- defined outside the constructor
- most frequent use cases
  - unique index (when working with databases)
  - counter
  - helper variable for some design patterns

Inheritance

- inheritance means that the class inherits part of its functionality from another class
- the class inheriting is the **subclass**; the class from which it inherits is the **superclass**
- using inheritance we can create a hierarchy, where we define ever more specific implementations of the class – subclasses will be more closely defined versions of the subclass
- in Python 2 you could create classes without a superclass, in Python 3 `object` is the default superclass
- Python supports multiple inheritance

Defining inheritance in Python

```
class Person:
    def __init__(self, name):
        self.name = name


class Student(Person):
    def __init__(self, name, year):
        self.name = name
        self.year = year
```

Calling superclass methods

- for supporting polymorphism and more efficient work with classes we should use as much the already defined functionality from superclasses
- keyword `super()`

```
class Person:
    def __init__(self, name):
        print('setting name in Person')
        self.name = name

class Student(Person):
    def __init__(self, name, year):
        super().__init__(name)
        self.year = year

john_doe = Student("John Doe", 1)
```

Multiple inheritance in Python

```
class Person: pass
class Worker(Person): pass
class Boss(Person): pass

class Manager(Worker, Boss): pass
class Secretary(Worker): pass
class BoardMember(Boss): pass

class CEO(Manager, BoardMember): pass
```

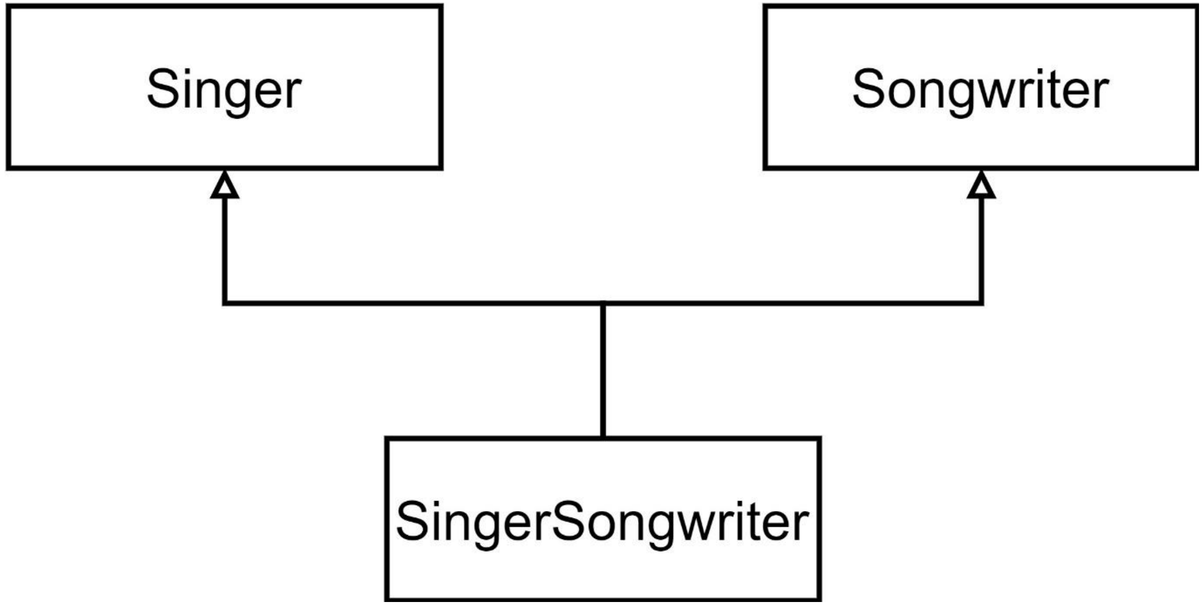
Problems with multiple inheritance

in class `Singer`:

```
sings = True
writes = False
```

in class `Songwriter`:

```
sings = False
writes = True
```

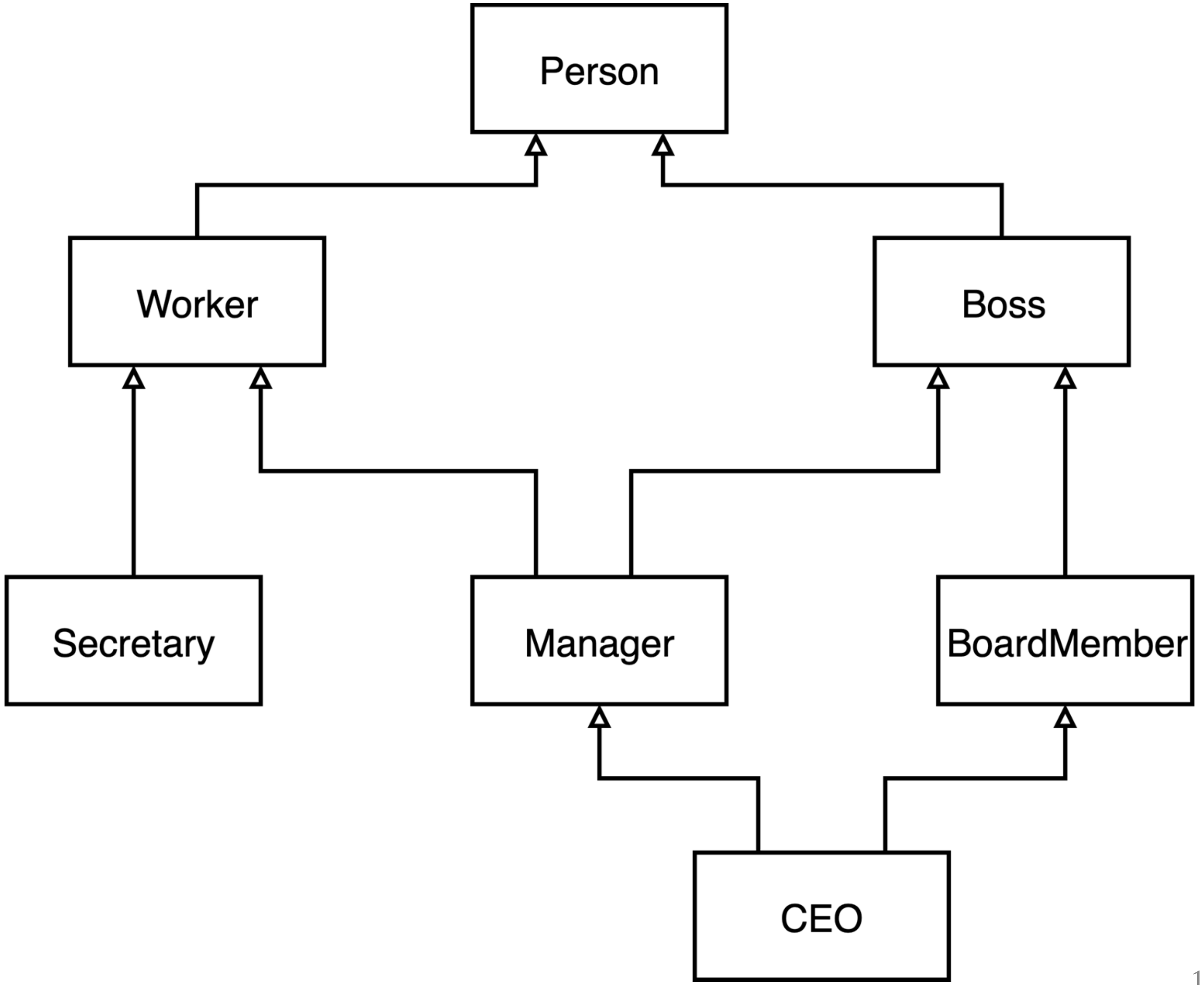


What values should `SingerSongwriter` inherit?

Name resolution with multiple inheritance

1. searching in the current class
2. deapth-first search in superclasses from left to right
3. we consider `object` last

- we can get the order using the `__mro__` field or calling `mro()`



Polymorphism

- polymorphism lets us use the same interface for different data types between which there is inheritance
- in reality this means that we can use objects of the subclass as if they were objects of the superclass
- considering that Python is dynamically typed, it is not that prominent

Main class metamethods in Python

- `__init__`
- `__str__`
- `__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, `__ge__`
- `__hash__`
- `__del__`
- `dir()`

Constructor

- method `__init__(self[, …])`
- defines the way in which we create a class instance
- called after starting the `__new__` method and “returns” a reference to the object
- the constructor of a subclass should call the constructor of the superclass

```
SuperClassName.__init__(args)
```

String representation of an object

- defined in `__str__(self)`
- used when calling
  - `str(my_object)`
  - `.format(my_object)`
  - `print(object)`
- a single return value of type `string`

Comparison methods

- used for comparing two objects
- functionality in different methods (not all necessary)

```
__eq__, __ne__, __lt__, __le__, __gt__, __ge__
  ==      !=      <      <=      >      >=
```

- return values are usually `True` or `False`, but could be any value

The `__hash__(self)` method

- used when calling the `hash()` function, or with operations with a hashed set of values (`set`, `frozenset`, `dictionary` – the key must be hashable)
- can return any value, the only requirement is that if `object1 == object2`, then `hash(object1) == hash(object2)`
- the implementation is closely connected to `__eq__`: the recommended approach is to store each value used to compare two objects into a tuple and call `hash()` over this tuple
- if you don't define `__eq__`, you shouldn't define `__hash__`; if you define `__eq__` but not `__hash__`, the class will represent an unhashable type

Finalizer

- defined in `__del__(self)`
- called in `del(my_object)`
- if a subclass defines `__del__`, it must call `__del__` from the superclass
- recommended when the object has active communication with a file or a database – we must close the communication channel
- within `__del__` it is possible to create a reference to the object to be deleted – resurrection
- errors generated in `__del__` are ignored, the message is only printed to `sys.stderr`

Function `dir()`

- used to get all attributes of an object or class
- for an object it returns a list of fields, methods, attributes of the class of which the object is an instance, and recursively each superclass's attributes
- for a class it returns a list of class variables, class methods, and recursively each superclass's attributes
- possible to define your own way of getting the list of attributes in `__dir__` but usually not necessary

Working with objects as numbers

- it is possible to define how to work with objects if they are arguments of simple arithmetic operations

```
+               object.__add__(self, other)
-               object.__sub__(self, other)
*               object.__mul__(self, other)
/               object.__truediv__(self, other)
//              object.__floordiv__(self, other)
%               object.__mod__(self, other)
divmod()        object.__divmod__(self, other)
**              object.__pow__(self, other)
```

Working with objects as numbers

```
+=                      object.__iadd__(self, other)
-=                      object.__isub__(self, other)
*=                      object.__imul__(self, other)
/=                      object.__itruediv__(self, other)
//=                     object.__ifloordiv__(self, other)
%=                      object.__imod__(self, other)
**=                     object.__ipow__(self, other)
abs()                   object.__abs__(self)
complex(object)         object.__complex__(self)
int(object)             object.__int__(self)
float(object)           object.__float__(self)
```

Rounding objects to integers

```
round()           object.__round__(self[, ndigits])
trunc()           object.__trunc__(self)
floor()           object.__floor__(self)
ceil()            object.__ceil__(self)
```

Example – class hierarchy

- defining a hierarchy of superclasses and subclasses
- inheritance and method overriding
- working with subclasses
- iterators

Method overriding in Python

- also called shadowing
- we specify a different functionality for subclasses, but we use the same name and parameters
- if the functionality is not completely different (which it shouldn't), we should use the implementation from the superclass

Method overloading in Python

- we have methods with the same name, but different return value/parameters
- typical example – overloading standard operations
- in Python we cannot overload user-defined methods, but can use default parameter values to the same effect

What do we want?

```
class Number:
    def __init__(self, value):
        self.num = value

    def multiply(self):
        return self.num * 2

    def multiply(self, number):
        return self.num * number
```

How do we solve it?

```
class Number:
    def __init__(self, value):
        self.num = value

    def multiply(self, number=2):
        return self.num * number
```

What do we want?

```
class Number:
    def __init__(self, value):
        self.num = value

    def multiply(self):
        return self.num * self.num

    def multiply(self, number):
        return self.num * number
```

How do we solve it?

```
class Number:
    def __init__(self, value):
        self.num = value

    def multiply(self, number=None):
        if number is None:
            return self.num * self.num
        else:
            return self.num * number
```

or use a method if a different name

Using subclasses

- since subclasses should be closer specifications of the superclass, the functionality of inherited classes should be similar
- therefore it is a good idea to use the functionality from the superclass (calling `super`)
- in Python two possibilities:
  - when we know the superclass's name (usual case):

    ```
    SuperClass.method_name(self, parameters)
    ```
  - when we don't know the superclass's name (creating a subclass from a library):

    ```
    super(SubClass, self).method_name(parameters)
    ```

Iterators

- for collections
- supporting `for` loops
- defined with two methods
  - `__iter__(self)`
    - creating the iterator
    - initializes helper variables
    - returns `self`
  - `__next__(self)`
    - returns the next element in the collection
    - once it reaches the end of the collection, it generates `StopIteration`

UML diagrams

- UML - Unified Modeling Language
- used for visualizing an object-oriented solution’s structure
- the basis for object-oriented modelling is the class diagram
- one block represents one class
- can also be used for modelling data

Class diagram

name

attributes

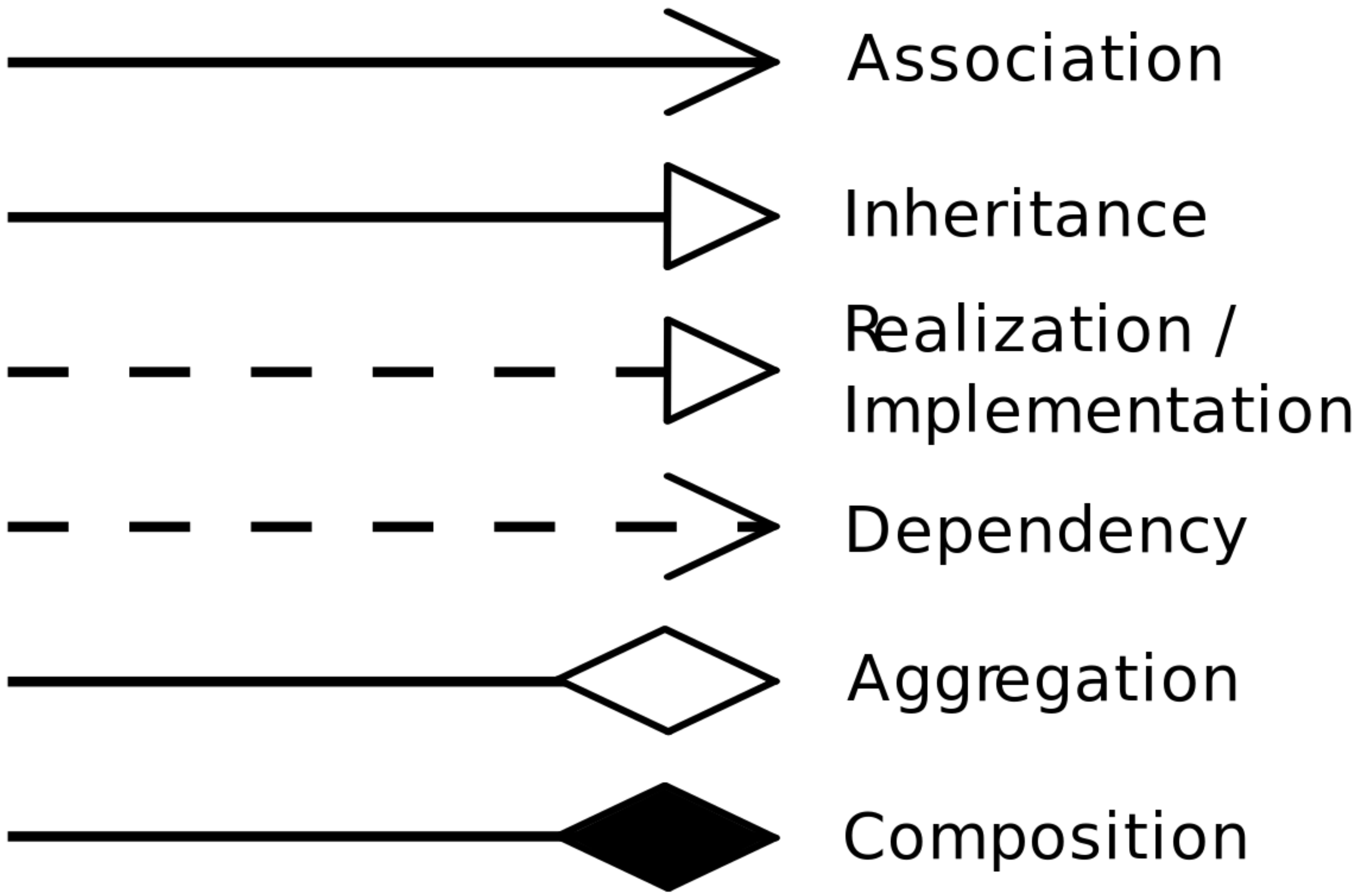
methods

Guitarist
family_name: string first_name: string idNum: int nextIdNum: int
__init__(familyName: string, firstName: string) getIdNum(): int __str__(): string __eq__(other: Guitarist): boolean

Encapsulation

- in Python we don’t have mutators, defined directly in the field name
  - `idNum: int` - public field
  - `_idNum: int` - private field
- class variables are underlined
  - <u>`nextIdNum: int`</u>

Relations between classes

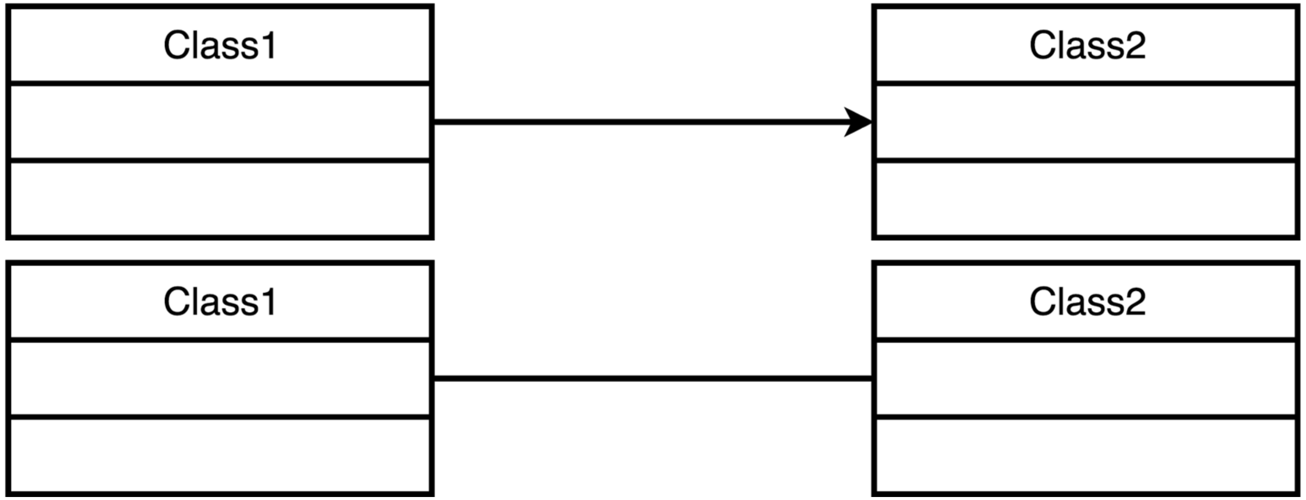


Cardinality

- for each relation we can defined cardinality (how many objects join in it)
  - **0** - 0
  - **0..1** - 0 or 1
  - **0..*/*** - 0 to n
  - **1/1..1** - 1
  - **1..*** - 1 to n
- shown at the end of the arrow next to the class

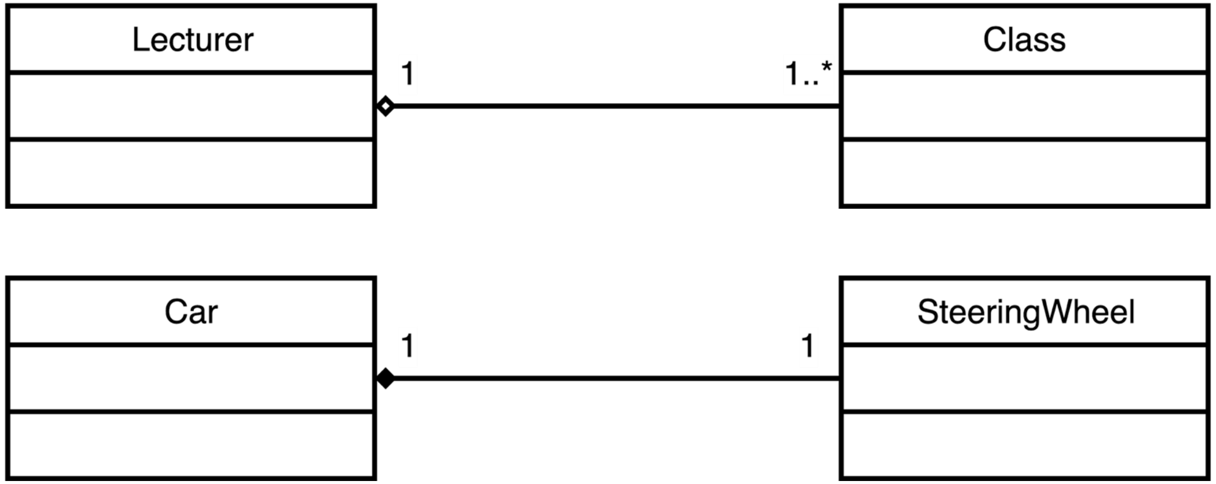
Association

- instance-level
- represented with an arrow (one-way) or a line (two-way)
- the object of a class uses a method of a different object
- one object uses another object
- one object is the attribute of another object



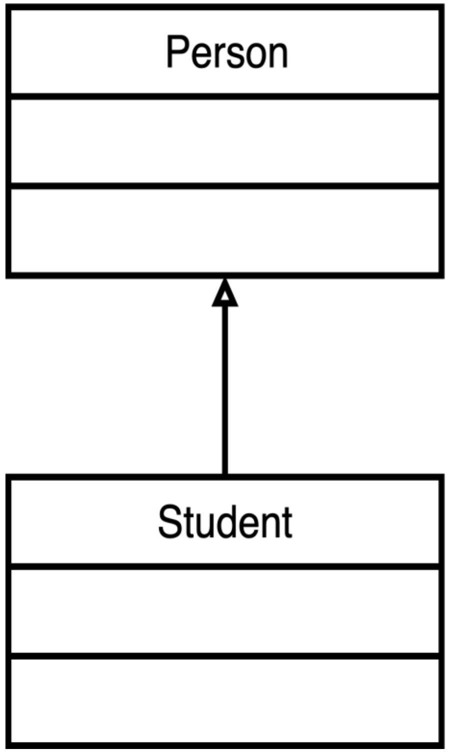
Aggregation and composition

- instance-level
- representing cases where the object of a class consists of/contains objects of another class
- aggregation – if we delete the container, individual parts can still exist
- composition – individual parts have no existence outside of the composite



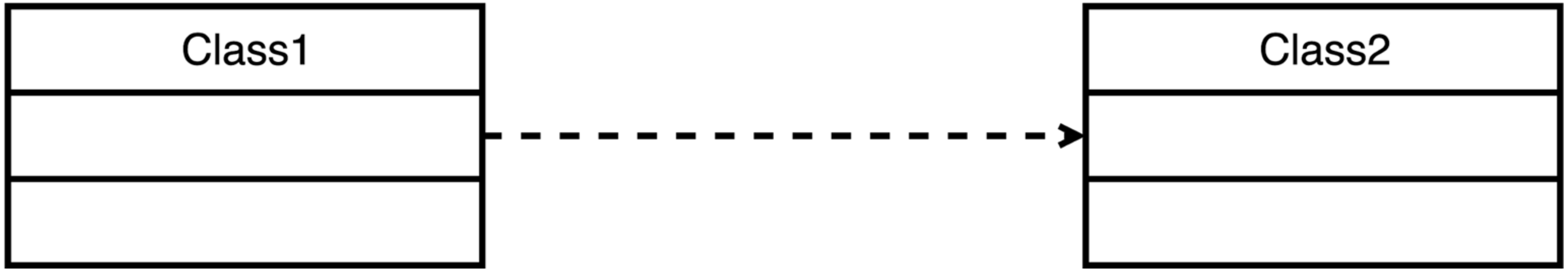
Inheritance

- class-level
- representing that the subclass is a closer specification of the superclass



Dependency

- general relation
- representing cases when one object uses another object, but the relation is much weaker than association
- the dependent class contains a method where the object of the independent class is a parameter



Conclusion

- abstraction
- encapsulation
- object state
- fields and class variables
- inheritance
- polymorphism
- constructor
- metamethods and their meaning
- comparing objects
- overriding and overloading
- UML diagrams, relations between classes