Introduction to OOP 2
Basic OOP concepts

Course: Object Oriented Programming (OOP)
CTU, FS, U12110
Matouš Cejnek

Contents

- Basic concepts of OOP
- Abstract classes, Interface
- Override and overload

What do we know about objects?

Class - template (blueprint) for objects

Object / instance - item created according to a class

Programmer writes code for classes, program creates instances and use them.

What do we now about OOP building blocks?

- **Classes**
- **Objects** (instances of classes)
- **Methods** (functions, procedures)
- **Attributes** (fields, members, variables, or properties)

What do we know about magic methods?

Magic methods are special methods. Implementation vary according to the language. In general:

- They have specific reserved namespace
- These methods are often called when some conditions are met
- Prime example is constructor and destructor

Basic concepts of OOP

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

Encapsulation

Basic concept of OOP

Encapsulation

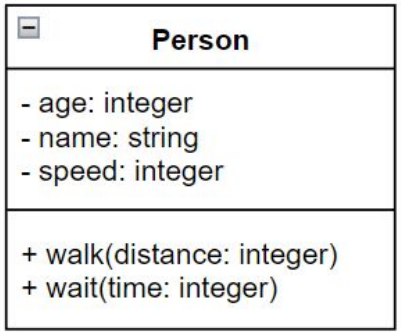
- **Encapsulation is data hiding**
- Encapsulation refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components.
- In most of the OOP languages, you can specify private, protected and public properties of an object.

Example of encapsulation

All person related properties are bundled in **Person class**

All attributes in this example are **private** (**-** in UML)

All methods in this example are **public** (**+** in UML)



Abstraction

Basic concept of OOP

CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Abstraction

- **Abstraction is implementation hiding**
- Abstraction is concept of hiding unnecessary details to allow focus on a greater picture.
- Abstraction is related to generalization (using same/similar concepts to handle different objects etc.)

Abstraction

- Abstraction is concept of hiding implementation details.
  For example we call:
  ```
  measured_distance = distance(-3)
  ```
  without knowing how the distance is implemented.
- Abstraction can be understand as generalization.
- Abstraction always uses encapsulation.

Abstraction vs encapsulation

Alternative explanation:

- Encapsulation is data hiding.
- Abstraction is implementation hiding.

Inheritance

Basic concept of OOP

Inheritance

Inheritance is concept of creating subclass from parent class in order to re-use some or all features of the parent class.

Types of inheritance

- **Class-based** programming is performed via defining classes of objects (this is the popular model of OOP).
- **Prototype-based** programming is performed via a process of reusing existing objects that serve as prototypes.

Class-based programming

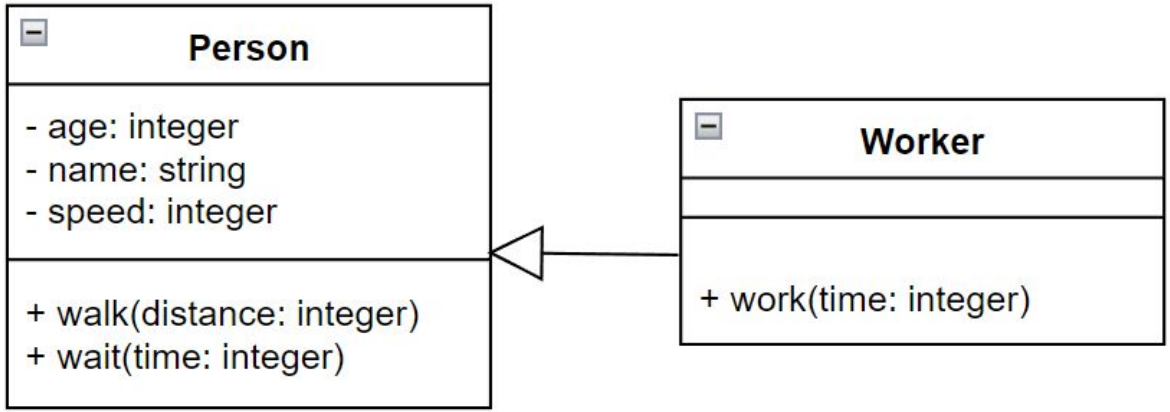
- The structure and behavior of an object are defined by a class, which is a blueprint of all specific type objects.
- An object must be explicitly created based on a class and an object thus created is considered to be an instance of that class.

Prototype-based programming

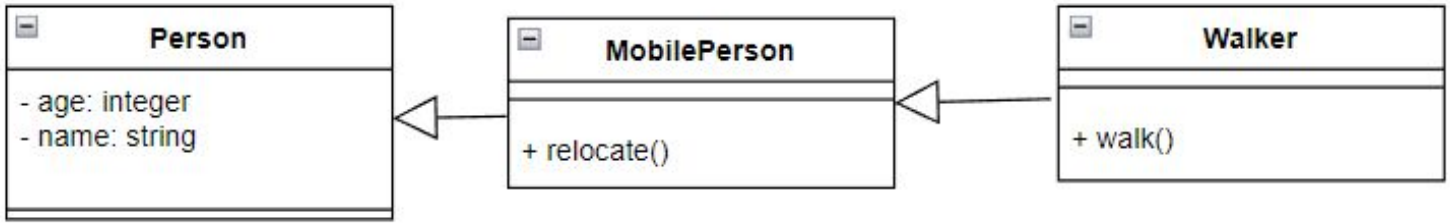
This model can also be known as prototypal, prototype-oriented, classless, or instance-based programming.

Example of inheritance

Class **Worker** inherit attributes and functions from more general **Person**

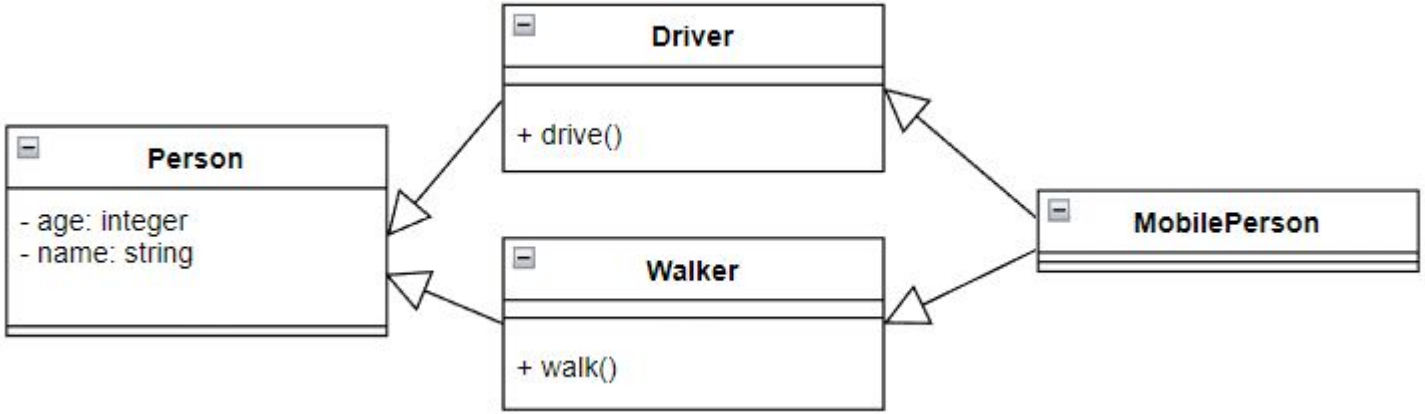


Example of inheritance



Example of multiple inheritance

Not every language allows complicated inheritance like below:



Polymorphism

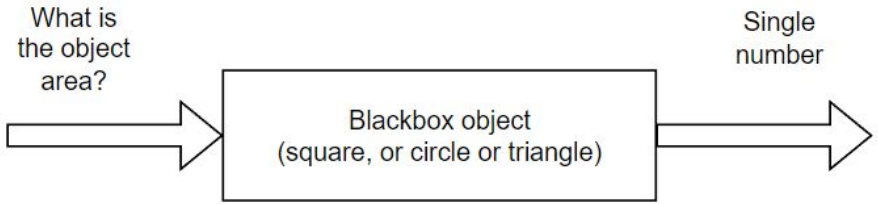
Basic concept of OOP

Polymorphism

Polymorphism is the provision of a single interface to entities of different types.

Polymorphism

Example: Imagine classes representing different shapes: circle, square, triangle. Even though they area is calculated in a different way, they all can return a number when area is requested from outside.



CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Polymorphism

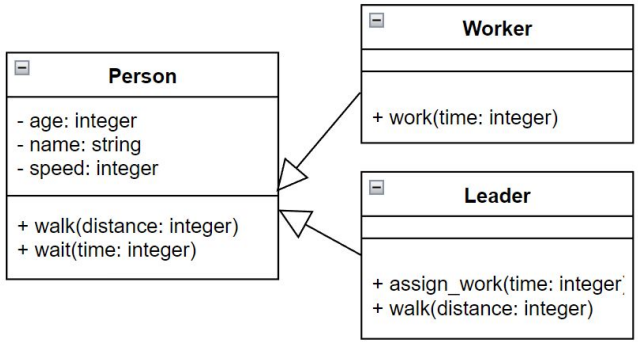
- **Polymorphism** is the provision of a single **interface** to entities of different types.
- In some OOP languages, the **interface** is used to define an abstract type that contains no data but defines behaviours as **method signatures.**
- The **method signature** is the combination of the method name and the parameter list.

Example of polymorphism

Leader and worker - they both can walk and wait (but it is not necessarily yield the same result).

```
# person
def walk(self, distance):
    return distance * self.speed

# leader
def walk(self, distance):
    return distance * 0.9 * self.speed
```



Interface, abstract class and metaclass

Abstract class and interface

- Both work as a template for standard classes.
- Proper multiple inheritance and duck-typing make them less needed.
- Implementation is strongly language specific.

Abstract class

- Abstract class is a special type of class that cannot be instantiated.
- Abstract class is designed to be inherited by subclasses that **either implement or override its methods**.
- It can contain abstract methods, or standard methods.

Abstract method

- It is a method without implementation.
- They might be used in abstract classes.

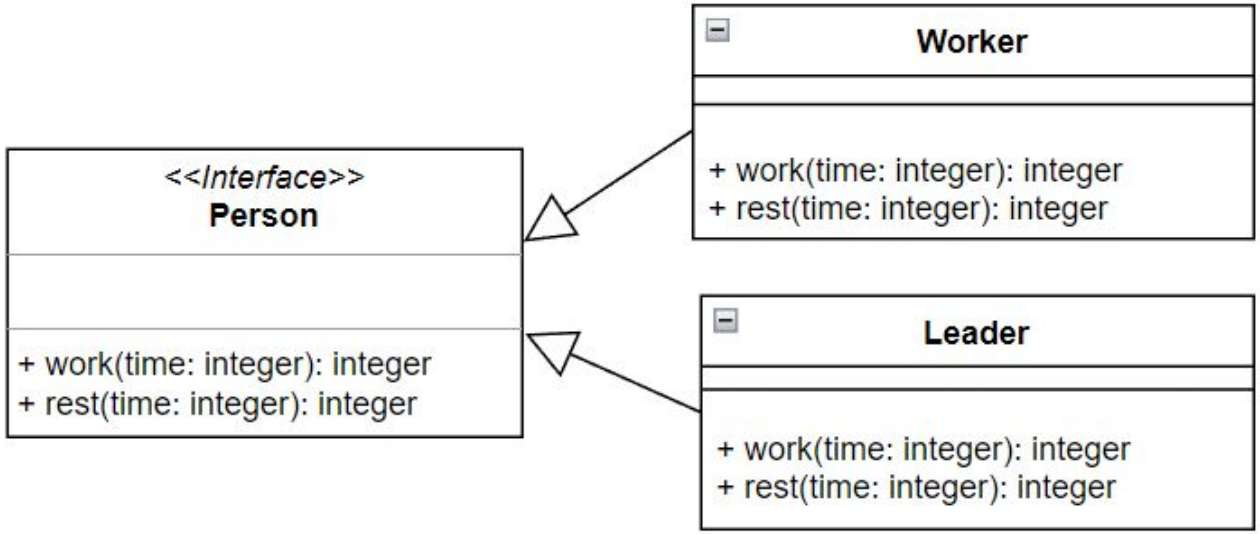
Abstract class

- Class with any abstract method is an abstract class
- Abstract class can have non-abstract method.

Interface

- It is an abstract type that is used to describe a behavior that classes must implement.
- It doesn’t have any implementation.
- Interface can contain only method declarations.

Interface example



Metaclass

- Metaclass is a class whose instances are classes.
- It allows us to create classes dynamically.
- Implementation is language specific.

“Metaclasses are deeper magic than 99% of users should ever worry about. ***If you wonder whether you need them, you don’t.****”*

— Tim Peters

Override and overload

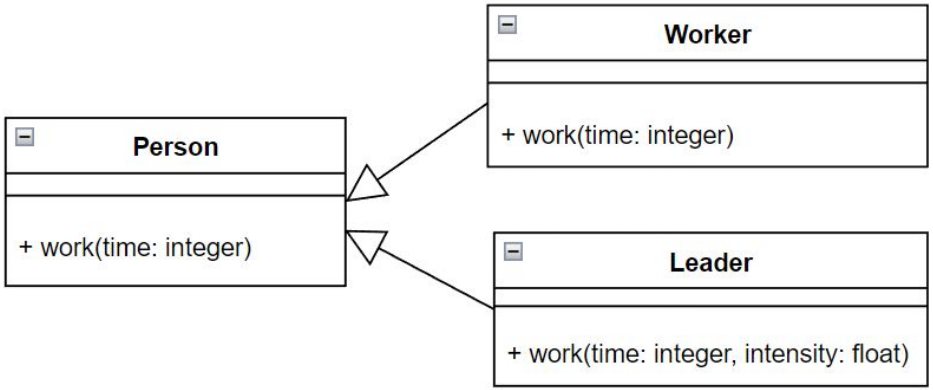
CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Method override and overload

- Overload - multiple functions with the same name for different cases (different input parameters)
- Override - replacement of a function with different function (but with same name)

Example of override

Inherited function *work* is replaced with different implementations.

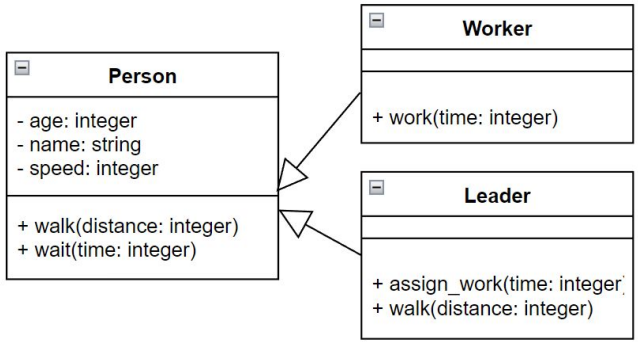


Example of override

Inherited function *walk* is replaced with different implementation.

```
# person
def walk(self, distance):
    return distance * self.speed

# leader
def walk(self, distance):
    return distance * 0.9 * self.speed
```

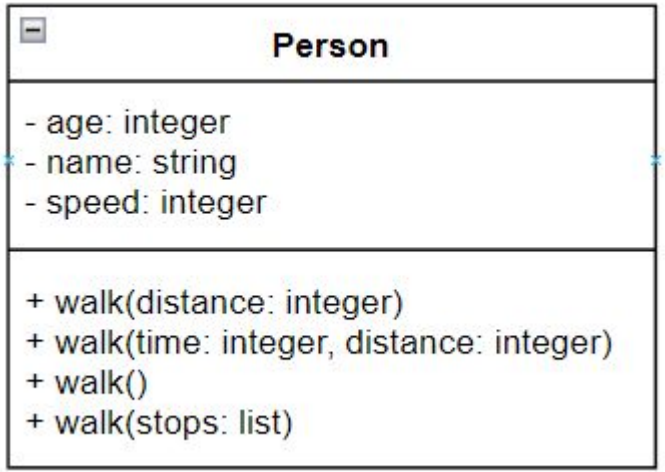


Method overload

- Overload allows us to have different implementation of the “same” functions for different parameters.
- Implementation is language (type system) specific.
- Conventional method overload is not available in Python.

Overload example

One class with multiple functions of the same name. The correct functions is chosen according provided arguments.



Overload example

C++ (same name, different parameters, correct function is used)

```
void add(int a, int b)
{
  cout << "sum = " << (a + b);
}
void add(double a, double b)
{
  cout << endl << "sum = " << (a + b);
}
```

False overload example

Python (if else workaround)

```
def add(a, b):
    if condition_related_to_inputs:
        # do something
    else:
        # do something else
```

Bonus content

Operator overload in Python

It is not possible to do C++ like overload in Python. However, you can overload (override) magic functions and operators.

```
class Example:
    def __init__(self, X):
        self.X = X
    def __add__(self, U):
        return self.X + U.X
result = Example(5) + Example(4)
```

Decorators in Python

```
def my_decorator(func):
    def wrapper(name):
        print("Decorating...")
        func(name)
        print("Decorated")
    return wrapper

@my_decorator
def my_function(name):
    print(name)

my_function("ABC")
```

Output:

```
Decorating...
ABC
Decorated
```

Decorators in Python (without @)

```
def my_decorator(func):
    def wrapper(name):
        print("Decorating...")
        func(name)
        print("Decorated")
    return wrapper

def my_function(name):
    print(name)

my_decorator(my_function)("ABC")
```

Output:

```
Decorating...
ABC
Decorated
```

Metaclasses in Python

We can create a class dynamically (**Example**):

```
class ExampleParent():
    pass


Example = type('Example', (ExampleParent,), {'value': 1})
e = Example()
```

Metaclasses in Python

Metaclass is always used:

```
class Example(metaclass=type):
    pass

e = Example()
```

Metaclasses in Python

We can use our own metaclass:

```
class MetaExample(type):
    pass

Example = MetaExample('Example', (), {'value': 1} )

e = Example()
```