Technische Universität Braunschweig

Institut für rechnergestützte Modellierung im Bauingenieurwesen

Algorithmen & Programmieren

Vorlesung 5

David Anton, M.Sc.

Pull out your calendar

Assignment 1

Available on StudIP:	30.11.2022
Submission Deadline:	16.12.2022

Assignment 2

Available on StudIP:	11.01.2023
Submission Deadline:	27.01.2023

Oral assessement: 06.02. - 10.02.2023

Succesfull completion of both assignments & the oral assessment is necessary to pass the module Algorithms & Progamming and to get the 8 ECTS!

Update of the knowledge base for your self-study

Technische Universität Braunschweig | Modulhandbuch: Master Computational Sciences in Engineering (CSE) (PO 2019)

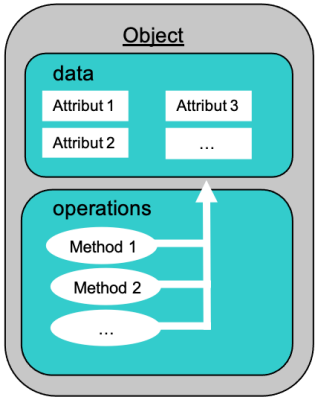
Modulbezeichnung: **Algorithms & Programming (Lab)**				Modulnummer: **BAU-STD5-48**	
Institution: Studiendekanat Bauingenieurwesen 5				Modulabkürzung:	
Workload:	240 h	Präsenzzeit:	56 h	Semester:	1
Leistungspunkte:	8	Selbststudium:	184 h	Anzahl Semester:	1
Pflichtform:	Pflicht			SWS:	4
Lehrveranstaltungen/Oberthemen: Algorithms & Programming (VÜ)					

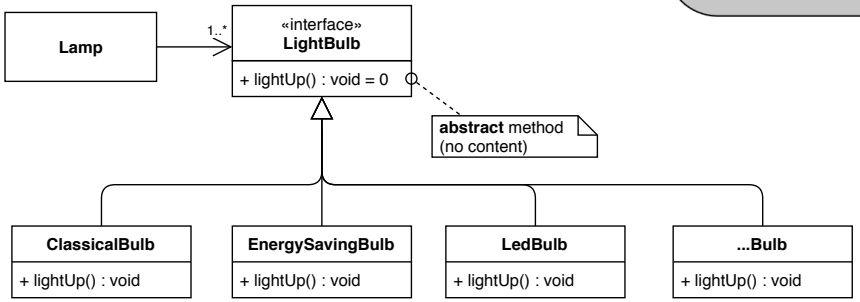
https://irmb.gitlab-pages.rz.tu-bs.de/knowledge-base/content/intro.html

Recapitulation: OOP - Principles

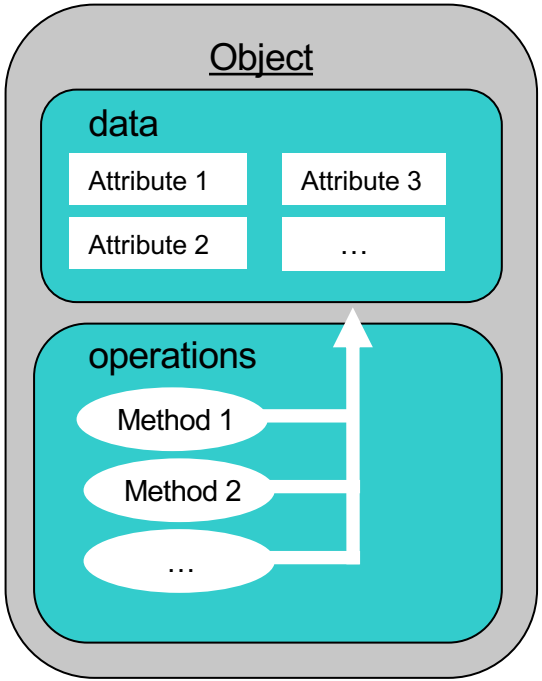
An object oriented programming language should at least support the following **three principles**:

- …
- …
- …



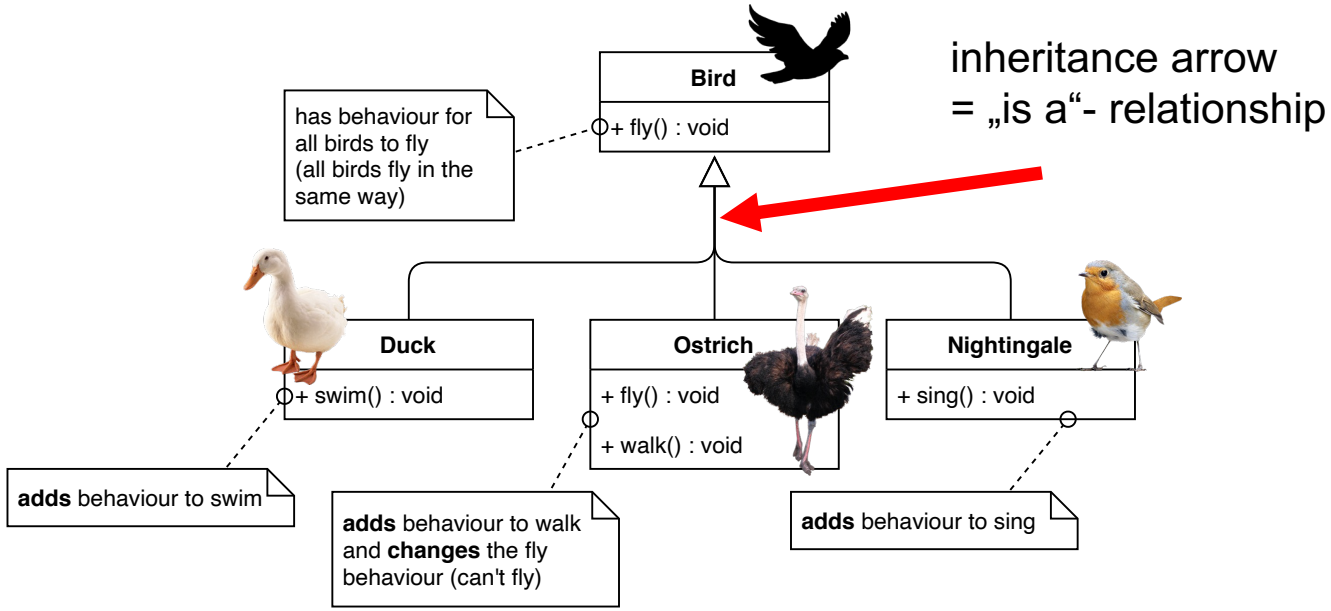


Recapitulation: OOP – Data encapsulation

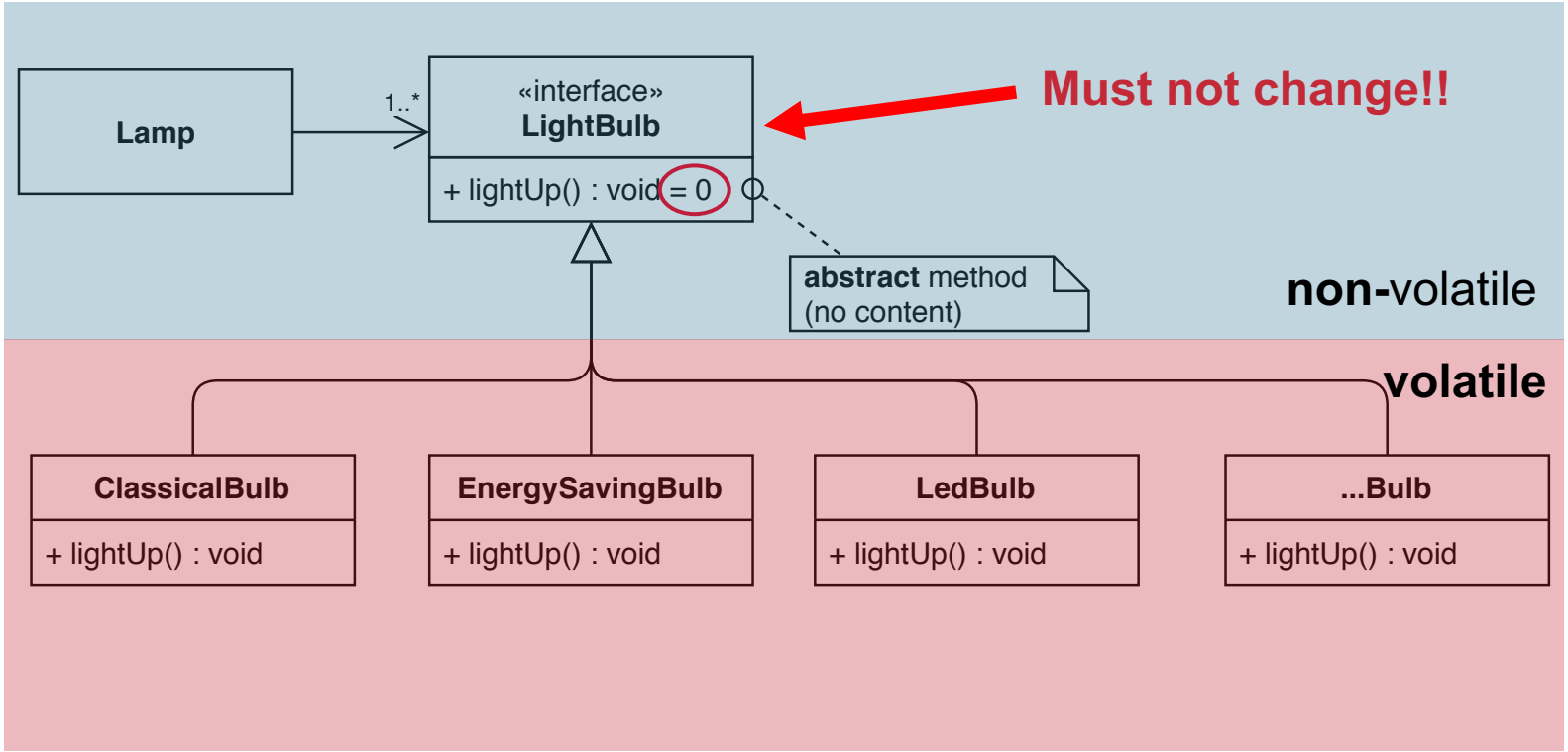


Remember the example where the radius of a circle should be set to a negative value.

Recapitulation: OOP - Inheritance



Recapitulation: OOP - Polymorphism



Classes

Class:

- A class is a data type for similar objects in an application program.
- The definition of the class is application specific (context sensitive properties).
- The class specifies the attributes and methods for objects and their access rights.

Object:

- An object is a variable with a defined class as data type.
- An object is an instance of a class.
- Each object has its own attributes and methods.
- The access rights are valid for all objects of the class.

Classes and objects

Class Definition:

- A class defines the variability and functionality from objects with similar features.
- The definition consists of the keyword class, a class name and class block, where the attributes and methods are defined.
- Each class definition is a type definition for objects (→ data type)

```python
class BankAccount:
    def __init__(
        self,
        first_name: str,
        last_name: str,
        iban: int,
        pin: int,
    ) -> None:
        self._first_name = first_name
        self._last_name = last_name
        self._iban = iban
        self._pin = pin
        self._balance = 0.0

    def deposit(self, amount: float) -> None:
        self._balance = self._balance + amount
        print(f"Deposited {amount}€")

    def pay_out(self, amount: float) -> float:
        self._balance = self._balance - amount
        print(f"Paid out {amount}€")
        return amount

    def __str__(self) -> str:
        return "{} {} IBAN:{} PIN:{} Balance:{}".format(
            self._first_name,
            self._last_name,
            self._iban,
            self._pin,
            self._balance,
        )
```

Classes and objects

Object creation:

- Each object is created as an instance of a class.
- A specific function (constructor) of the class is called each time an object is created.
- Deleting of objects is done automatically if there is no further reference to them due to Python‘s „garbage collector“.

```python
account = BankAccount("John", "Doe", 123456789, 1234)
```

```python
class BankAccount:
    def __init__(
        self,
        first_name: str,
        last_name: str,
        iban: int,
        pin: int,
    ) -> None:
        self._first_name = first_name
        self._last_name = last_name
        self._iban = iban
        self._pin = pin
        self._balance = 0.0

    def deposit(self, amount: float) -> None:
        self._balance = self._balance + amount
        print(f"Deposited {amount}€")

    def pay_out(self, amount: float) -> float:
        self._balance = self._balance - amount
        print(f"Paid out {amount}€")
        return amount

    def __str__(self) -> str:
        return "{} {} IBAN:{} PIN:{} Balance:{}".format(
            self._first_name,
            self._last_name,
            self._iban,
            self._pin,
            self._balance,
        )
```

Attributes

Attributes:

- Attributes are defined in the constructor.
- If an object is created (constructor is called), the attributes are initialized.
- A defined attribute is called via the object.

Static attributes:

- Static attributes are defined on class level.
- **All** objects of a class have the same static attributes.
- A static attribute is called outside of the class via the class.

```python
class BankAccount:
    #static attribute
    iban_length = 9

    def __init__(
        self,
        first_name: str,
        last_name: str,
        iban: int,
        pin: int,
    ) -> None:
        # Usually rather candidate for private attribute ...
        self.iban = iban
        self._first_name = first_name
        self._last_name = last_name
        self._pin = pin
        self._balance = 0.0

    # Methods definitions ...


if __name__ == "__main__":
    #Access static attribute (So far no object is created!)
    print("IBAN length: {}".format(BankAccount.iban_length))
    account = BankAccount("John", "Doe", 123456789, 1234)
    #Access attribute
    print("IBAN: {}".format(account.iban))
```

Methods

A method is an instruction which is applied to a set of parameters resulting in potential modification of these. The method‘s definition is part of a class.

Definition:

name(parameter list) -> return type:

command block

```
1  def pay_out(self, amount: float) -> float:
2      self._balance = self._balance - amount
3      print(f"Paid out {amount}€")
4      return amount
```

Signature: can not identity the function by parameter, that means cannot function overloading, defined by name only

The signature of a method is composed of the method name and the parameter list.

But: In Python, the method is identified based on the method name only. Python does not allow overloading of methods. Methods with the same name are overwritten.

Methods

Reference “self”:

A method is called via an existing object. The object is an implicit parameter of the method and can be called with the reference **self.**

Static methods:

- Static methods are defined using the **@staticmethod** decorator.
- A static method is called outside the class definition via the class.
- No access to attributes from instances of the class.

```python
class BankAccount:
    def __init__(
        self,
        first_name: str,
        last_name: str,
        iban: int,
        pin: int,
    ) -> None:
        self._first_name = first_name
        self._last_name = last_name
        self._iban = iban
        self._pin = pin
        self._balance = 0.0

    @staticmethod
    def print_iban_info() -> None:
        print("IBAN is 9 digits long.")

    def deposit(self, amount: float) -> None:
        self._balance = self._balance + amount
        print(f"Deposited {amount}€")

    # other methods ...


if __name__ == "__main__":
    BankAccount.print_iban_info()
    account = BankAccount("John", "Doe", 123456789, 1234)
    account.deposit(100)
```

Methods

That the main wont execute while importing other class by other files all line will be compiled but only things define will be imported

Arguments:

- If there are no parameters defined, the argument list is empty.
- The arguments of a method are separated by commas.
- An argument is a constant, a variable or an expression.

- The number (and data types) of the arguments for a method call must be the same number (and data types) as the parameter of the method definition.

rem: can add data member after definition
e.g. calculator.diu = 1 (valid)

```python
class Calculator:
    def __init__(self) -> None:
        pass

    def sum(self, a: int, b: int) -> int:
        print("Sum two integers.")
        return a + b

    def sum(self, a: int, b: int, c: int) -> int:
        print("Sum three integers.")
        return a + b + c


if __name__ == "__main__":
    calculator = Calculator()
    calculator.sum(1, 2)
```

What output do you expect?

Access rights

In classical object oriented programming languages (e.g. Java, C++), the access to attributes and methods of objects is controlled by access rights:

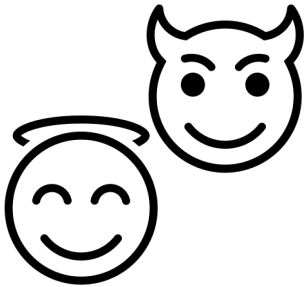
- **public:** Access is allowed from any point in the program.
- **protected:** The definition depends on the language. In Java, e.g., access is allowed from every point in the package where the class is defined and in subclasses which expand the class.
- **private:** Access is denied from every point outside the class definition.

What do you think access rights might be good for?

Access rights

In contrast, Python has no built-in access rights. In Python, the access rights *public* and *private* are realized by convention only:

- The names of *private* attributes and methods begin with an underscore.
- Attributes and methods whose names do not begin with an underscore are *public*.
- The access right *protected* does not exist (not even by convention).



Note that in Python, private attributes and methods are only private by convention. This does not mean that you cannot still access the attributes and methods from outside the class. However, the underscore suggests not to do that. This requires discipline!

Constructor

Constructor:

- The constructor is a specific method.
- The method name of the constructor is *__init__*.
- In the constructor, the attributes for an instance of the class are initialized.
- The constructor has an implicit return type (returns the created object).

```python
class BankAccount:
    def __init__(
        self,
        first_name: str,
        last_name: str,
        iban: int,
        pin: int,
    ) -> None:
        self._first_name = first_name
        self._last_name = last_name
        self._iban = iban
        self._pin = pin
        self._balance = 0.0
```

Instantiation and access

Instantiation:

- *object_name* = ClassType(argument list)
- *object _name* holds the reference to the created object

return pointer pointing at the data piece of object

Access to attributes:

- Inside a class definition: *self.attribute_name*
- Outside a class definition: *object _name.attribute_name*

Access to methods:

- Inside a class definition: *self.method_name(parameter_list)*
- Outside a class definition: *object_name.method_name(parameter_list)*

Variables are not boxes

Variables: a pointer to the mem data piece

Variables in Python are not boxes. Variables hold a reference to an object (almost everything in Python is an object) or are assigned to an object.

```python
def demo_variables() -> None:
    a = [0, 1, 2]
    b = a
    a.append(3)
    print(b)
    print("ID of a: {}".format(id(a)))
    print("ID of b: {}".format(id(b)))
```

Note that a new object (with a new reference) is created if one tries to modify an immutable object. This applies, for instance, to augmented assignments (e.g., a=a+1).

```python
def demo_immutable_types() -> None:
    c = 1
    d = c
    print("ID of c: {}".format(id(c)))
    print("ID of d: {}".format(id(d)))
    c = c + 1
    print(d)
    print("ID of c: {}".format(id(c)))
    print("ID of d: {}".format(id(d)))
```

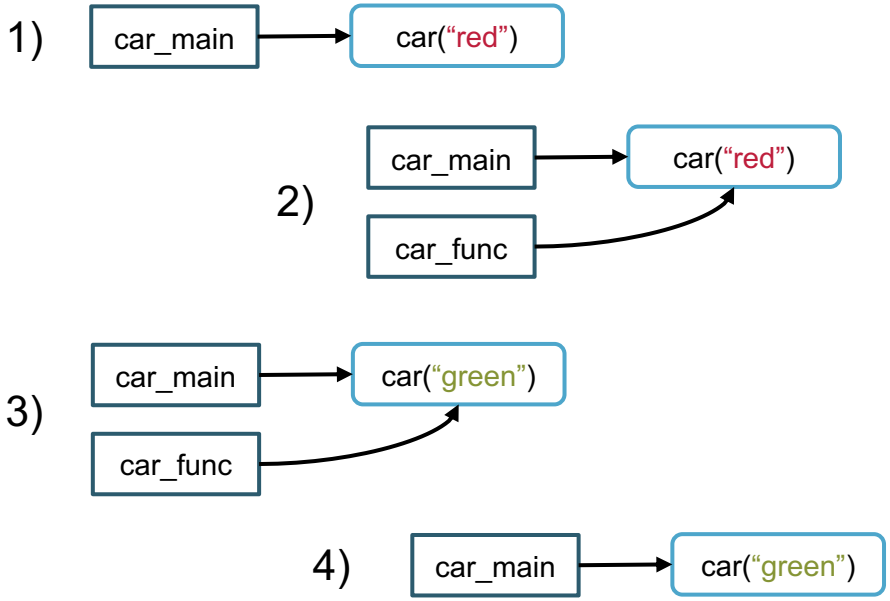
a new data piece after the computation

Call-by-object-reference

- In Python, parameters are passed in the mode *call-by-object-reference* (aka *call-by-sharing*). According to this mode, each parameter in a function gets a copy of the reference hold by the corresponding parameter from the parameter list. As a result, the parameter inside the function becomes an alias of the passed parameter.
- By giving the functions references to the passed objects, the functions may modify mutable objects, but can not change their identity.
- When passing immutable types (str, int, float, tuple) the mode *call-by-object-reference* behaves like *call-by-value*. Whenever an immutable object is attempted to be modified inside the function, a new object is created. As a result, the object referenced from the variable outside the function stays unmodified.

Call-by-object-reference

Memory assignments:

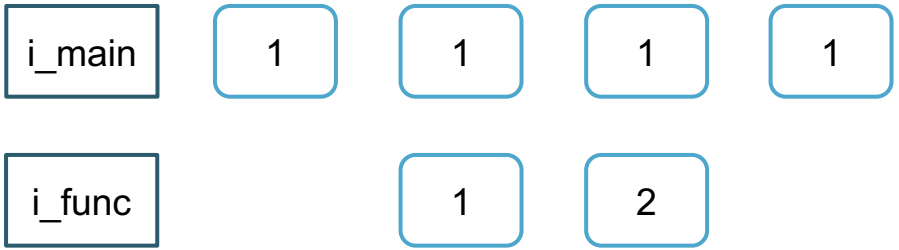



```python
class Car():
    def __init__(self, color: str) -> None:
        self.color = color

    def __str__(self) -> str:
        return "The color of the car is {}".format(self.color)

def change_color(car_func: Car) -> None:
    car_func.color = "green"

def demo() -> None:
    car_main = Car("red")
    print(car_main)
    change_color(car_main)
    print(car_main)
```

object is user defined data class, so mutable
so here is pass by reference

Call-by-object-reference

Memory assignments:




```python
def add_one(i_func: int) -> int:
    i_func = i_func + 1
    return i_func

def demo_immutable_types() -> None:
    i_main = 1
    print(i_main)
    add_one(i_main)
    print(i_main)
```

Inheritance

Inheritance / Class families

- A class family is a hierarchy of classes which is created according to a parent-children-relation.
- The essential feature of a class family is the inheritance principle.
- Methods and attributes of a parent class are handed to the child class.
- The child class can have own attributes and methods as well as overwrite methods from parent classes. It is an **extension and modification** of the parent class.
- The principle of inheritance can be used to write **non-redundant code, enable polymorphism** and to reduce the programming effort.

Superclass: Inherit attributes / methods to all subclasses.

Subclasses: Inherit all attributes and methods of its superclasses, they can overwrite them, and they can define additional attributes / methods.

Inheritance

Crane

- payload: float
- boom_length: float

+ lift_load(): void

```python
class TruckCrane(Crane):
    def __init__(
        self,
        payload: float,
        boom_length: float,
        speed: float
    ) -> None:
        super().__init__(payload, boom_length)
        self._speed = speed

    def drive(self) -> None:
        print("The truck crane drives.")
```

TowerCrane

- tower_height: float

+ turn_tower(): void

TruckCrane

- speed: float

+ drive(): void

Multiple inheritance

Python supports multiple inheritance where the subclass inherits from more than one base classes.

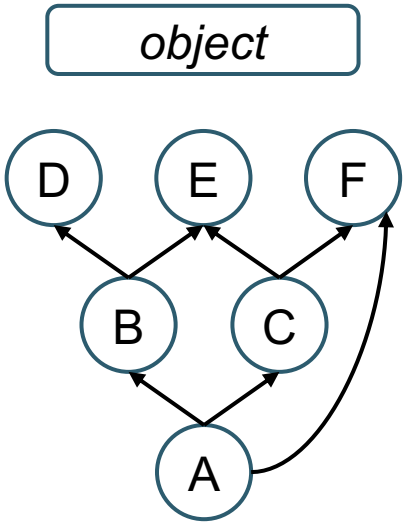
In this case, the subclass inherits all attributes and methods from its base classes. The attributes and methods are searched according to the **Method Resolution Order (MRO)**, which is defined for the most simplest cases as follows:

- The attributes and methods are first searched in the current class.
- If they are not found, the interpreter continues searching in the base classes in depth-first, left-right order without searching a class twice.

The MRO is slightly more complex for diamond relationships .

Note that in Python, every class inherits from *object* which is the root of the Python class hierarchy.

left first search



search the attribute and method by LFS

MRO: A, B, D, C, E, F

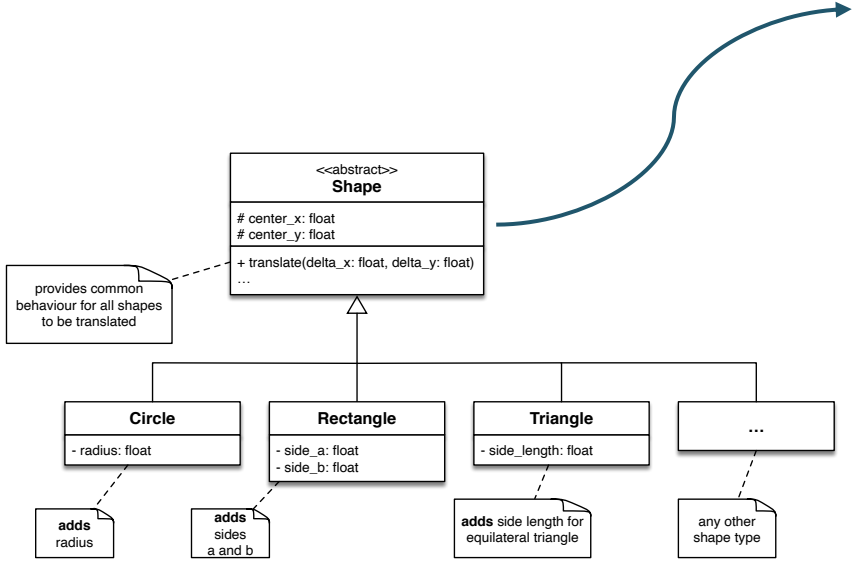
Abstract classes and methods

An *abstract class* is a class that is declared abstract - it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An *abstract method* is a method that is declared without an implementation. Implementing the abstract method is the responsibility of the subclass. If the subclass does not provide an implementation, the subclass is also an abstract class.

In Python, abstract classes and methods are implemented using the ***ABC*** class and the ***@abstractmethod*** decorator (see next slide).

Abstract classes and methods




```python
# Standard library imports
from abc import ABC, abstractmethod


class Shape(ABC):
    def __init__(self, center_x: float, center_y: float) -> None:
        self._center_x = center_x
        self._center_y = center_y

    def translate(self, delta_x: float, delta_y: float) -> None:
        self._center_x = self._center_x + delta_x
        self._center_y = self._center_y + delta_y

    @abstractmethod
    def calculate_area(self) -> float:
        raise NotImplementedError


if __name__ == "__main__":
    # TypeError: Can't instantiate abstract class Shape
    # with abstract method calculate_area
    shape = Shape(center_x=1.0, center_y=2.0)
```

Interfaces

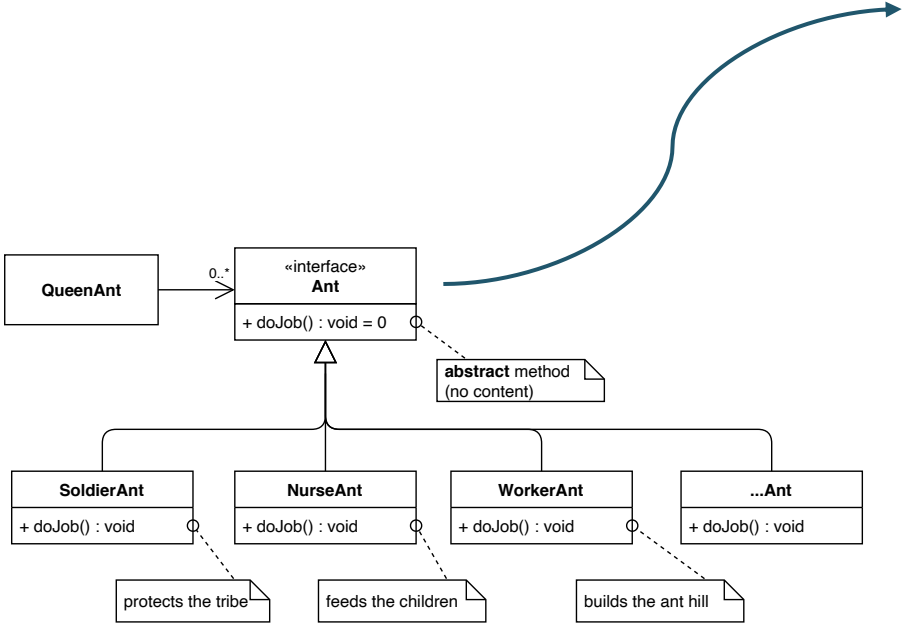
In Python, *protocol* describes the interface required by a caller.

Protocols are based on the principle *structural subtyping* or *duck-typing*. They only serve to write down the required interface. However, compared to abstract classes, the *protocol* is not inherited. With this, the *protocol* defines an implicit interface. Just like abstract classes, protocols cannot be instantiated.

A *protocol* is defined by inheriting the **Protocol** class from the *typing* package (see next slide).

Interfaces

define a protocol and see is
the class satisfied the protocol




```python
# Standard library imports
from typing import Protocol


class Ant(Protocol):
    def do_your_job(self) -> None:
        pass

class WorkerAnt():
    def do_your_job(self) -> None:
        print("Worker ant builds hill.")

class NurseAnt():
    def do_your_job(self) -> None:
        print("Nurse ant feeds children.")

class QueenAnt():
    def give_commands(self, ants: list[Ant]) -> None:
        for ant in ants:
            ant.do_your_job()


if __name__ == "__main__":
    queen = QueenAnt()
    ants = [WorkerAnt(), NurseAnt()]
    queen.give_commands(ants)
```

Abstract classes versus interfaces

- Unlike interfaces, abstract classes can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation. If an abstract class contains only abstract method declarations, it should be declared as an interface instead.

- By comparison, abstract classes are most commonly subclassed to share pieces of implementation. A single abstract class is subclassed by similar classes that have a lot in common (the implemented parts of the abstract class), but also have some differences (the abstract methods).

Overriding methods and attributes

A method in the subclass with the same method name as a method in the superclass overrides the superclass's method.

The same applies for attributes.

Within a subclass definition, the method in the superclass cannot be referenced by its simple name. Instead the method name must be accessed through *super().method_name()*.

```python
class Animal:
    def __str__(self) -> None:
        return "Animal"

class Cat(Animal):
    def __str__(self) -> None:
        return "Cat"


if __name__ == "__main__":
    cat = Cat()
    # Since Cat inherits from Animal, cat is of type Animal
    # (and of type Cat -> class hierarchy).
    print(isinstance(cat, Animal))
    print(cat)
```

Modules and packages

To make types easier to find and use and to avoid naming conflicts, programmers bundle groups of related types into *modulus* and *packages*.

In Python, a *module* is a file containing Python definitions and statements. The modules name is the file name and has the suffix *.py*. Definitions from a module can be imported to other modules and the main module.

Packages are used to organize Python modules. In Python, packages are realized by directories. For Python to treat a directory as package, a top-level *__init__.py* file is required.

For more information on modules and packages, please see the Python documentation:
https://docs.python.org/3/tutorial/modules.html

to state which module is available to other module
package is the modules located in same directory