Object Oriented Programming
Using
Python

Teemu Matilainen
teemu.matilainen@savonia.fi

Lecture 4#

- **Aggregation**
- **Composition**
- **Objects and references**
- **Objects as arguments to functions**
- **Objects as arguments to methods**
- **Instance of the class as an argument to a method**

Lecture 4#

Aggregation

Aggregation is a concept in object-oriented programming (OOP) that represents a "has-a" relationship between two classes. It is a form of association where one class contains an object of another class but does not control the lifecycle of that object. In simpler terms, it denotes a relationship where one class is part of another class.

Aggregation

In aggregation, the contained object can exist independently of the containing object. It doesn't imply ownership, and the contained object can be shared among multiple instances of the containing class or even exist outside of it.

```python
class Employee:
    def __init__(self, name, employee_id):
        self.name = name
        self.employee_id = employee_id

    def display_info(self):
        print(f"Employee {self.name} (ID: {self.employee_id})")

class Department:
    def __init__(self, name):
        self.name = name
        self.employees = []  # Aggregation: Department has a list of employees

    def add_employee(self, employee):
        self.employees.append(employee)

    def display_employees(self):
        print(f"Employees in the {self.name} department:")
        for employee in self.employees:
            employee.display_info()
```

```python
employee1 = Employee("name 1", 101)
employee2 = Employee("name 2", 102)
employee3 = Employee("name 3", 103)

hr_department = Department("HR")
hr_department.add_employee(employee1)
hr_department.add_employee(employee2)

engineering_department = Department("Engineering")
engineering_department.add_employee(employee3)

# Display employees in each department
hr_department.display_employees()
engineering_department.display_employees()
```

```
Employees in the HR department:
Employee name 1 (ID: 101)
Employee name 2 (ID: 102)
Employees in the Engineering department:
Employee name 3 (ID: 103)
```

Aggregation (Car)

Aggregation is different from **composition**, where the contained object is considered a part of the containing object, and its lifecycle is controlled by the containing object. In aggregation, the contained object is more independent and can be shared among multiple objects.

```python
class Engine:
    def __init__(self, fuel_type):
        self.fuel_type = fuel_type

    def start(self):
        print(f"The engine starts {self.fuel_type} fuel.")

    def stop(self):
        print("The engine stops.")

class Car:
    def __init__(self, make, model, engine):
        self.make = make
        self.model = model
        self.engine = engine  # Aggregation: Car has an Engine

    def start(self):
        print(f"The {self.make} {self.model} starts.")
        self.engine.start()

    def stop(self):
        print(f"The {self.make} {self.model} stops.")
        self.engine.stop()

my_engine = Engine("gasoline")
my_car = Car("Toyota", "Camry", my_engine)  # Aggregation: Car has an Engine

my_car.start()
my_car.stop()
```

```
The Toyota Camry starts.
The engine starts gasoline fuel.
The Toyota Camry stops.
The engine stops.
```

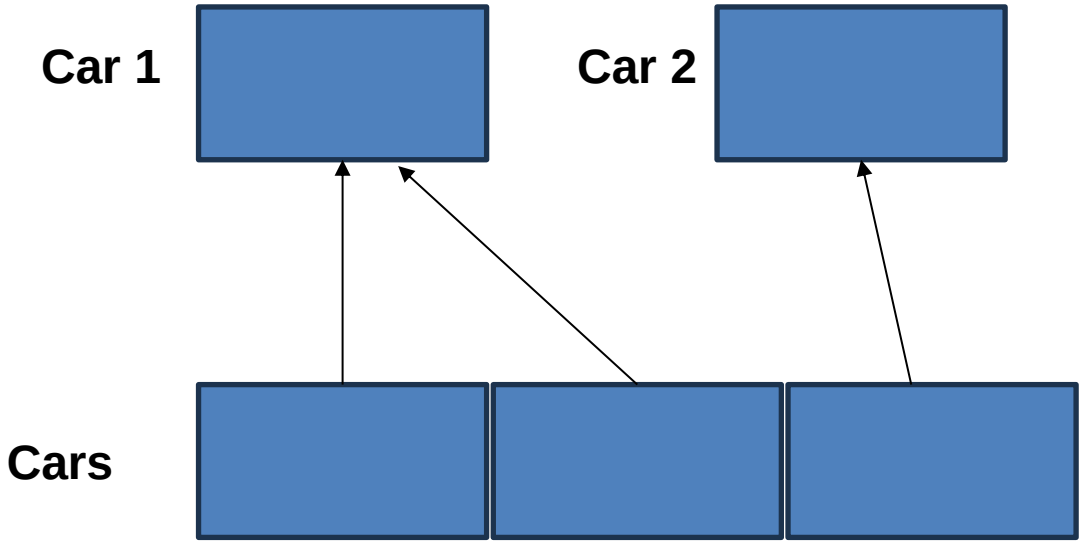
Composition:

When using **Composition**, the difference is that here we are not creating an object of the Engine inside the Car class, rather than that we are creating an object of the Engine outside and passing it as a parameter of Engine class which yields the same result.

```python
class Engine:
    def __init__(self, fuel_type):
        self.fuel_type = fuel_type

    def start(self):
        print(f"The engine starts running with {self.fuel_type} fuel.")

    def stop(self):
        print("The engine stops.")

class Car:
    def __init__(self, make, model, fuel_type):
        self.make = make
        self.model = model
        self.engine = Engine(fuel_type)  # Composition: Car has an Engine

    def start(self):
        print(f"The {self.make} {self.model} starts.")
        self.engine.start()  # Delegating the start operation to the Engine

    def stop(self):
        print(f"The {self.make} {self.model} stops.")
        self.engine.stop()   # Delegating the stop operation to the Engine

# Example usage
my_car = Car("Toyota", "Camry", "gasoline")
my_car.start()
my_car.stop()
```

```
The Toyota Camry starts.
The engine starts running with gasoline fuel.
The Toyota Camry stops.
The engine stops.
```

Objects and references



Let’s create a class Engine in its own file

```python
class Engine:
    def __init__(self, fuel_type):
        self.fuel_type = fuel_type

    def start(self):
        print(f"The engine starts running with {self.fuel_type} fuel.")

    def stop(self):
        print("The engine stops.")
```

Every value in Python is an object. Any object you create based on a class you've defined yourself works exactly the same as any "regular" Python object. For example, objects can be stored in a list:

```python
from engine import Engine

class Car:
    def __init__(self, make, model, engine):
        self.make = make
        self.model = model
        self.engine = engine

    def start(self):
        print(f"The {self.make} {self.model} starts.")
        self.engine.start()

    def stop(self):
        print(f"The {self.make} {self.model} stops.")
        self.engine.stop()

if __name__ == "__main__":
    cars = []

    my_first_engine = Engine("gasoline")
    my_second_engine = Engine("diesel")

    my_first_car = Car("Toyota", "Camry", my_first_engine)
    my_second_car = Car("Ford", "Escord", my_first_engine)
    my_third_car = Car("Mitsubishi", "Outlander", my_second_engine)

    cars.append(my_first_car)
    cars.append(my_second_car)
    cars.append(my_third_car)
```

```python
for car in cars:
    car.start()

for car in cars:
    car.stop()
```

The references at indexes 0, 3 and 4 in the list refer to the same object. Either one of the references can be used to access the object. The reference at index 1 and 2 refers to a different object, although with seemingly the same contents. Changing the contents of this latter object does not affect the other one.

Reference

```python
if __name__ == "__main__":
    cars = []

    my_first_engine = Engine("gasoline")
    my_second_engine = Engine("diesel")

    my_first_car = Car("Toyota", "Camry", my_first_engine)
    my_second_car = Car("Ford", "Escord", my_first_engine)
    my_third_car = Car("Mitsubishi", "Outlander", my_second_engine)

    cars.append(my_first_car)

    cars.append(my_second_car)
    cars.append(my_third_car)

    cars.append(my_first_car)
    cars.append(my_first_car)

    my_first_car.make = "Toyota new"
    for car in cars:
        car.start()

    for car in cars:
        car.stop()
```

```
The Toyota new Camry starts.
The engine starts running with gasoline fuel.
The Ford Escord starts.
The engine starts running with gasoline fuel.
The Mitsubishi Outlander starts.
The engine starts running with diesel fuel.
The Toyota new Camry starts.
The engine starts running with gasoline fuel.
The Toyota new Camry starts.
The engine starts running with gasoline fuel.
```

Dictionary usage for Objects:

- Let us used dictionary to store the object data structure.

```
if __name__ == "__main__":
    cars = {}

    my_first_engine = Engine("gasoline")
    my_second_engine = Engine("diesel")

    my_first_car = Car("Toyota", "Camry", my_first_engine)
    my_second_car = Car("Ford", "Escord", my_first_engine)
    my_third_car = Car("Mitsubishi", "Outlander", my_second_engine)

    cars["k1"] = my_first_car
    cars["k2"] = my_second_car
    cars["k3"] = my_first_car

    my_first_car.make = "Toyota new"

    for car in cars:
        cars[car].start()
```

```
The Toyota new Camry starts.
The engine starts running with gasoline fuel.
The Ford Escord starts.
The engine starts running with gasoline fuel.
The Toyota new Camry starts.
The engine starts running with gasoline fuel.
```

How can we test references?

The operator **is** been used for checking if the two references refer to the exact same **object**, while the operator **==** will tell you if the contents of the objects are the **same**.

```
print(cars[0] is cars[1])
print(cars[0] is cars[2])
print(cars[0] is cars[3])
print(cars[0] is cars[4])

print(cars[0] == cars[1])
print(cars[0] == cars[2])
print(cars[0] == cars[3])
print(cars[0] == cars[4])
```

```
False
False
True
True
False
False
True
True
```

Objects as arguments to functions

Let's have a look at a simple example where a function receives a reference to an object of type my_first_car as its argument. The function then changes the name of the car. Both the function and the main function calling it access the same object, so the change is apparent in the main function as well.

```python
from engine import Engine

class Car:
    def __init__(self, make, model, engine):
        self.make = make
        self.model = model
        self.engine = engine

    def start(self):
        print(f"The {self.make} {self.model} starts.")
        self.engine.start()

    def stop(self):
        print(f"The {self.make} {self.model} stops.")
        self.engine.stop()

def change_name(car: Car):
    car.make = "new name"
```

```python
change_name(my_first_car)
```

```
The Toyota Camry starts.
The engine starts running with gasoline fuel.
The new name Camry starts.
The engine starts running with gasoline fuel.
```

Objects within function

It is also possible to create objects within functions. If a function returns a reference to the newly created object, it is also accessible within the main function!!!!!

```python
from random import randint, choice
from engine import Engine

class Car:
    def __init__(self, make, model, engine):
        self.make = make
        self.model = model
        self.engine = engine

    def start(self):
        print(f"The {self.make} {self.model} starts.")
        self.engine.start()

    def stop(self):
        print(f"The {self.make} {self.model} stops.")
        self.engine.stop()

def new_car():
    make = ["Toyota","Ford","Mitsubishi"]
    model = ["Camry", "Escord", "Outlander"]

    # randomly car name
    ma = choice(make)
    mo = choice(model)
    eng = Engine(str(randint(10000,99999)))

    return Car(ma, mo, eng)
```

```python
if __name__ == "__main__":
    cars = []

    cars.append(new_car())
    cars.append(new_car())
    cars.append(new_car())

    for car in cars:
        car.start()

    for car in cars:
        car.stop()
```

```
The Mitsubishi Outlander starts.
The engine starts running with 93048 fuel.
The Mitsubishi Outlander starts.
The engine starts running with 75624 fuel.
The Ford Camry starts.
The engine starts running with 62587 fuel.
The Mitsubishi Outlander stops.
The engine stops.
The Mitsubishi Outlander stops.
The engine stops.
The Ford Camry stops.
The engine stops.
```

Objects as arguments to methods

Objects can act as arguments to methods.

Car Class contains a is_the_same method what compares car object and another car object engines (given as an argument)

```python
from engine import Engine

class Car:
    def __init__(self, make, model, engine):
        self.make = make
        self.model = model
        self.engine = engine

    def start(self):
        print(f"The {self.make} {self.model} starts.")
        self.engine.start()

    def stop(self):
        print(f"The {self.make} {self.model} stops.")
        self.engine.stop()

    def is_the_same_type(self, e2: Engine):
        return self.engine is e2

if __name__ == "__main__":

    my_first_engine = Engine("gasoline")
    my_second_engine = Engine("diesel")

    my_first_car = Car("Toyota", "Camry", my_first_engine)
    my_second_car = Car("Ford", "Escord", my_first_engine)

    print(my_first_car.is_the_same_type(my_second_car.engine))
    print(my_first_car.is_the_same_type(my_second_engine))
```

```
True
False
```

Instance of the class as an argument to a method

Let's assume we want to write a program which compares car objects. We can write a separate function for this purpose

```python
from engine import Engine

class Car:
    def __init__(self, make, model, engine):
        self.make = make
        self.model = model
        self.engine = engine

    def start(self):
        print(f"The {self.make} {self.model} starts.")
        self.engine.start()

    def stop(self):
        print(f"The {self.make} {self.model} stops.")
        self.engine.stop()

def is_the_same_type(c1: Car, c2: Car):
    return c1 is c2

if __name__ == "__main__":

    my_first_engine = Engine("gasoline")
    my_second_engine = Engine("diesel")

    my_first_car = Car("Toyota", "Camry", my_first_engine)
    my_second_car = Car("Ford", "Escord", my_first_engine)
    my_third_car = Car("Mitsubishi", "Outlander", my_second_engine)

    print(is_the_same_type(my_first_car, my_first_car))
    print(is_the_same_type(my_first_car, my_third_car))
```

```
True
False
```

One of the principles of object-oriented programming is to include any functionality which handles objects of a certain type in the class definition, as methods. So instead of a function we could write a method which allows us to compare the car with another car object

Here the car object which the method is called on is referred to as self, while the other car object is c1.

Remember, calling a method differs from calling a function. A method is attached to an object with the **quote notation: “Car”**

```python
from engine import Engine

class Car:
    def __init__(self, make, model, engine):
        self.make = make
        self.model = model
        self.engine = engine

    def start(self):
        print(f"The {self.make} {self.model} starts.")
        self.engine.start()

    def stop(self):
        print(f"The {self.make} {self.model} stops.")
        self.engine.stop()

    def is_the_same_type(self, c1: "Car"):
        return self is c1

if __name__ == "__main__":

    my_first_engine = Engine("gasoline")
    my_second_engine = Engine("diesel")

    my_first_car = Car("Toyota", "Camry", my_first_engine)
    my_second_car = Car("Ford", "Escord", my_first_engine)
    my_third_car = Car("Mitsubishi", "Outlander", my_second_engine)

    print(my_first_car.is_the_same_type(my_first_car))
    print(my_first_car.is_the_same_type(my_third_car))
```

```
True
False
```

Is there any other way to compare objects?

```python
if __name__ == "__main__":

    my_first_engine = Engine("gasoline")
    my_second_engine = Engine("diesel")

    my_first_car = Car("Toyota", "Camry", my_first_engine)
    my_second_car = Car("Ford", "Escord", my_first_engine)
    my_third_car = Car("Mitsubishi", "Outlander", my_second_engine)

    print(my_first_car.is_the_same_type(my_first_car))
    print(my_first_car.is_the_same_type(my_third_car))

    print(my_first_car)
    print(my_third_car)
    print(str(my_third_car) == str(my_third_car))
```

```
True
False
<__main__.Car object at 0x0000020EBBB9EF90>
<__main__.Car object at 0x0000020EBBB9F3B0>
True
```

Assignment 6#

Secret Society with Artifacts

Create a class named Artifact with the following attributes:

name (string): the name of the artifact.
power (string): the power or ability associated with the artifact.

Implement methods within the Artifact class:

__init__(self, name, power): A constructor method that initializes the name and power of the artifact.

Create a class named Member with the following attributes:

secret_identity (string): a unique identifier for the member.
name (string): the public name of the member.
access_level (string): the level of access the member has within the society.
artifacts (list): a list to store instances of the Artifact class representing artifacts owned by the member.

Implement methods within the Member class:

__init__(self, secret_identity, name, access_level): A constructor method that initializes the secret_identity, name, and access_level of the member.

add_artifact(self, artifact): A method that takes an Artifact object as an argument and adds it to the member's list of artifacts.

Create a class named Society with the following attributes:

society_name (string): the name of the secret society.
members (list): a list to store instances of the Member class representing society members.

Implement methods within the Society class:

__init__(self, society_name): A constructor method that initializes the society_name and an empty list for members.

add_member(self, member): A method that takes a Member object as an argument and adds the member to the society.

remove_member(self, member): A method that takes a Member object as an argument and removes the member from the society.

print_society_info(self): A method that prints information about the society, including the society name, a list of members, and their associated artifacts.

Create instances of the Artifact, Member, and Society classes, add members to the society, associate artifacts with members, and print information for both members and the society

You can copy the test code from here:

```
# Create instances of the Artifact, Member, and Society classes
artifact1 = Artifact("Invisibility Cloak", "Camouflage")
artifact2 = Artifact("Mind Control Amulet", "Persuasion")

member1 = Member("S001", "Agent Smith", "High")
member2 = Member("S002", "Agent Brown", "Medium")

# Add artifacts to members
member1.add_artifact(artifact1)
member2.add_artifact(artifact2)

# Create an instance of the Society class
secret_society = Society("The Illuminati")

# Add members to the society
secret_society.add_member(member1)
secret_society.add_member(member2)

# Print information for each member and the society
secret_society.print_society_info()
```

```
# Create instances of the Artifact, Member, and Society classes
artifact1 = Artifact("Invisibility Cloak", "Camouflage")
artifact2 = Artifact("Mind Control Amulet", "Persuasion")

member1 = Member("S001", "Agent Smith", "High")
member2 = Member("S002", "Agent Brown", "Medium")

# Add artifacts to members
member1.add_artifact(artifact1)
member2.add_artifact(artifact2)

# Create an instance of the Society class
secret_society = Society("The Illuminati")

# Add members to the society
secret_society.add_member(member1)
secret_society.add_member(member2)

# Print information for each member and the society
secret_society.print_society_info()
```

```
Society: The Illuminati
Members:
  Agent Smith (Secret ID: S001, Access Level: High)
  Artifacts:
    Invisibility Cloak (Power: Camouflage)
  Agent Brown (Secret ID: S002, Access Level: Medium)
  Artifacts:
    Mind Control Amulet (Power: Persuasion)
```