ENG 346
Data Structures and Algorithms for Artificial Intelligence
Object-Oriented Programming

Dr. Mehmet PEKMEZCİ

mpekmezci@gtu.edu.tr

https://github.com/mehmetpekmezci/GTU-ENG-346

ENG-346-FALL-2025 Teams code is **0uv7jlm**

Agenda

- Object-Oriented Basics
- Class Definitions
- Encapsulation
- Inheritance
- Polymorphism
- Operator Overloading
- Examples

What is a Class?

- A blueprint or template in object-oriented programming that defines the *structure* and *behavior* of objects.

- An abstraction to a key concept through its structure and behavior.

- Attributes: Data the class will be holding.
- Methods: Operations on the data

- Class encapsulates both data (attributes/properties) and behavior (methods/messages) making it easier to model and manipulate complex systems.

Example: Car Class

Class: Car
Attributes: • Color • Production Year • Maximum Speed • …
Methods: • Accelerate() • Decelerate() • Turn_on_the_lights() • …

What is an Object?

- An object is an *instance* of a class.
  - Realization of a Class

- Objects represent real-world entities, concepts, or things in your program.

Example

GEBZE TECHNICAL UNIVERSITY

CarFile.py

```
class CarClass :
    def __init__(self, color, production_year,maximum_speed):
        self.color=color
        self.production_year=production_year
        self.maximum_speed=maximum_speed
        self.current_speed=0
        self.lights_status=False

    def accelerate(self,velocity):
        if self.current_speed >= self.maximum_speed:
           print(f"""Already passed the maximum speed {self.maximum_speed}.
                 Our speed is {self.current_speed}""")
        else:
           self.current_speed+=velocity
           print(f"Accelerated current velocity is :{self.current_speed}")

    def decelerate(self,velocity):
        if self.current_speed <= 0:
           print(f"Speed is already 0.")
        else:
           self.current_speed-=velocity
           print(f"Decelerated current velocity is :{self.current_speed}")

    def set_lights(self,lights_status):
        self.lights_status=lights_status
        if self.lights_status == True:
           print(f"Lights are ON")
        else:
           print(f"Lights are OFF")
```

main.py

```
#!/usr/bin/python3

from CarFile import CarClass

def main():
  carObject_1=CarClass("BLUE","1956",120)
  carObject_2=CarClass("BLUE","1924",50)

  carObject_1.accelerate(50)
  carObject_1.accelerate(50)
  carObject_1.accelerate(50)

  carObject_1.decelerate(50)
  carObject_1.decelerate(50)
  carObject_1.decelerate(50)
  carObject_1.decelerate(50)

  carObject_2.accelerate(50)
  carObject_2.accelerate(50)

if __name__ == '__main__':
 main()
```

```
mpekmezci@cobalt:~/workspace/GTU-ENG-346-PRIVATE/object_oriented_programming-codes$ python3 main.py
Accelerated current velocity is :50
Accelerated current velocity is :100
Accelerated current velocity is :150
Decelerated current velocity is :100
Decelerated current velocity is :50
Decelerated current velocity is :0
Speed is already 0.
Accelerated current velocity is :50
Already passed the maximum speed 50.
                 Our speed is 50
```

Example: My Car

Class: Car
Attributes: • Color • Production Year • Maximum Speed • …
Methods: • Accelerate() • Decelerate() • Turn_on_the_lights() • …

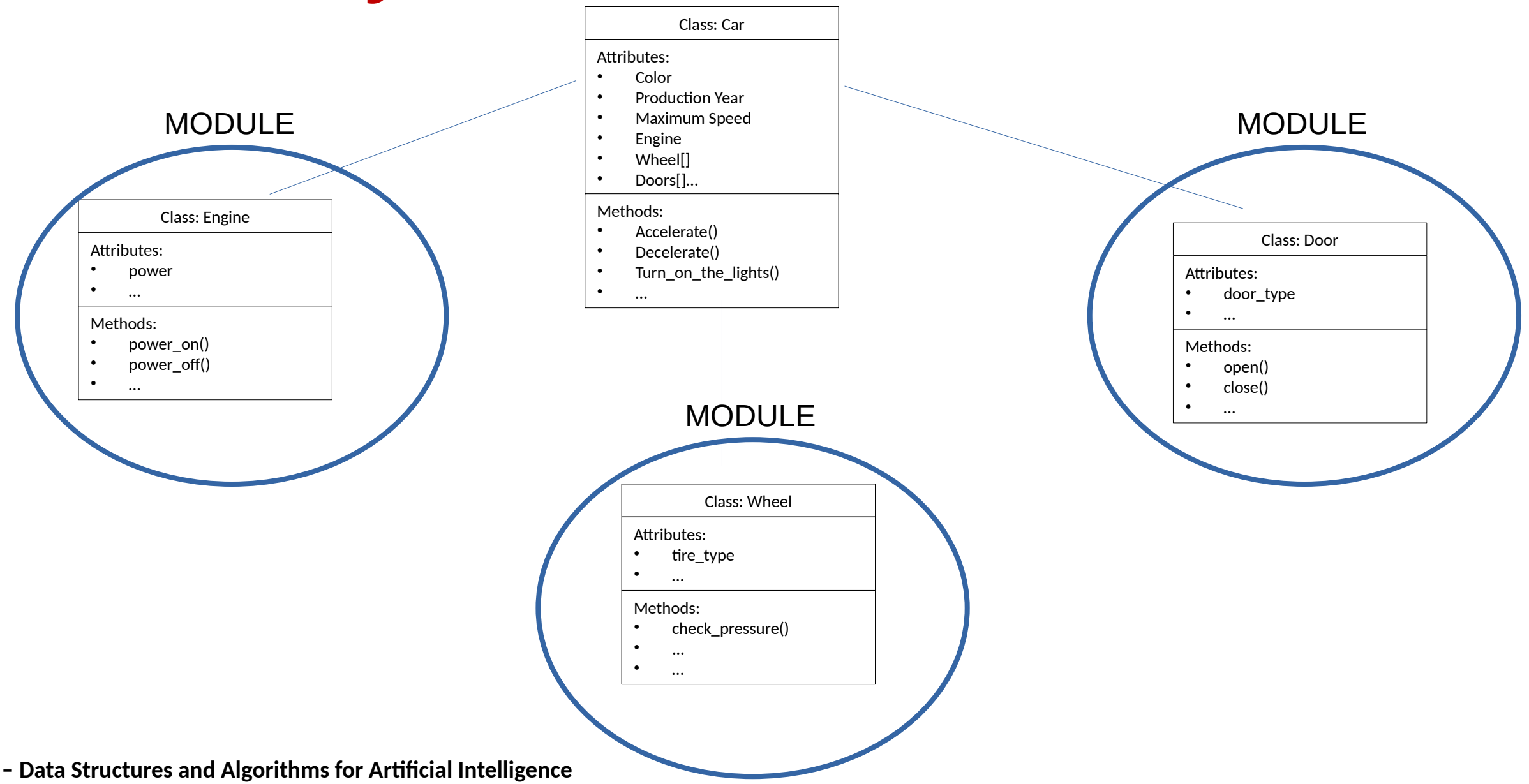
Object: My Car
Attributes: • Color: Black • Production Year: 2010 • Maximum Speed: 180 • …
Methods: • Accelerate() • Decelerate() • Turn_on_the_lights() • …

Object: Wife's Car
Attributes: • Color: White • Production Year: 2015 • Maximum Speed: 170 • …
Methods: • Accelerate() • Decelerate() • Turn_on_the_lights() • …

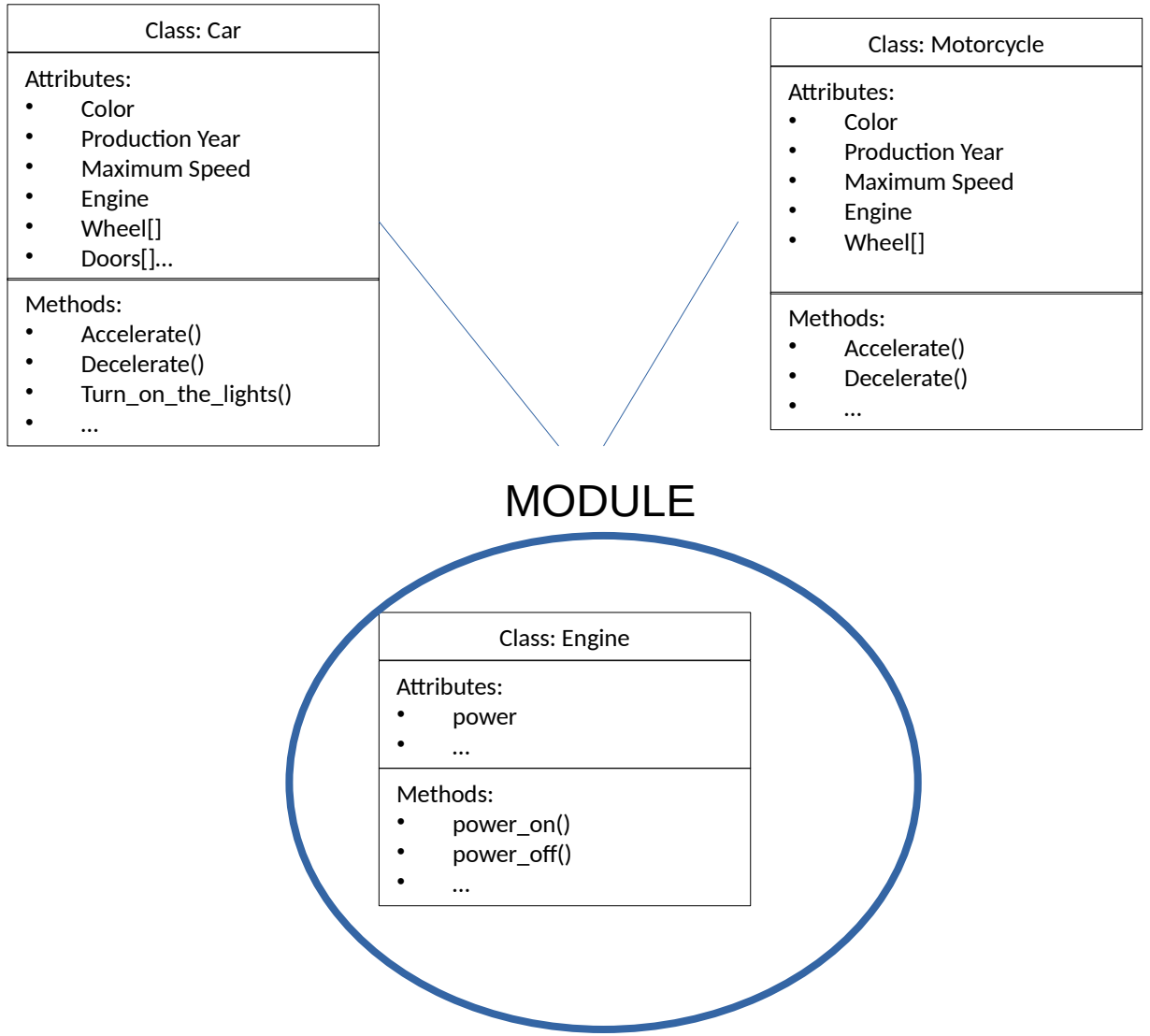
Benefits of OOP

- Modularity: Helps to organize the code. Example : a house or apartment can be viewed as consisting of several interacting units; electrical, heating, cooling, plumbing, structure, etc.
- Reusability: Encourages code reuse at the Class level.
- Abstraction: Abstraction is the process of hiding internal implementation details and showing only essential functionality to the user. It focuses on what an object does rather than how it does it.
- Encapsulation: Bundling data (attributes) and the methods (functions) that operate on that data into a single unit.
- Inheritance: Inherit attributes and methods from ancestors (parent classes).
- Polymorphism: Treat subclasses as if they are of one type, the Superclass.
- Scalability: Perform coding on large scale.
- Ease of Maintenance: Code organization and modularity makes it easy to debug, localize bugs, and perform maintenance.
- Team Collaboration: Work on individual classes without disturbing one another.
- Real-World Modeling: Natural way of thinking, design, and programming.

Modularity


Class: Car
Attributes:
• Color
• Production Year
• Maximum Speed
• Engine
• Wheel[]
• Doors[]...
Methods:
• Accelerate()
• Decelerate()
• Turn_on_the_lights()
• ...
MODULE
Class: Engine
Attributes:
• power
• ...
Methods:
• power_on()
• power_off()
• ...
MODULE
Class: Door
Attributes:
• door_type
• ...
Methods:
• open()
• close()
• ...
MODULE
Class: Wheel
Attributes:
• tire_type
• ...
Methods:
• check_pressure()
• ...
• ...

Reusability



Abstraction

Class: Car
Attributes: • Color • Production Year • Maximum Speed • Engine • Wheel[] • Doors[]…
Methods: • **Public** Accelerate() • **Public** Decelerate() • **Public** Turn_on_the_lights() • **Private __check_speed()**

```
class CarClass :
    def __init__(self, color, production_year,maximum_speed):
        self.color=color
        self.production_year=production_year
        self.maximum_speed=maximum_speed
        self.current_speed=0.1
        self.lights_status=False

    def accelerate(self,velocity):
        if  self.__check_speed(self.current_speed+velocity):
            self.current_speed+=velocity
            print(f"Accelerated current velocity is :{self.current_speed}")

    def decelerate(self,velocity):
        if  self.__check_speed(self.current_speed-velocity):
            self.current_speed-=velocity
            print(f"Decelerated current velocity is :{self.current_speed}")

    def __check_speed(self,targetSpeed):
        if targetSpeed <= 0:
            print(f"Speed is already 0.")
            return False
        if targetSpeed >= self.maximum_speed:
            print(f"""Already passed the maximum speed {self.maximum_speed}.
                  Our speed is {self.current_speed}""")
            return False
        return True

    def set_lights(self,lights_status):
        self.lights_status=lights_status
        if self.lights_status == True:
            print(f"Lights are ON")
        else:
            print(f"Lights are OFF")
~
```

Abstraction

```
#!/usr/bin/python3

from CarFileAbstraction import CarClass

def main():
  carObject_1=CarClass("BLUE","1956",120)
  carObject_2=CarClass("BLUE","1924",50)

  carObject_1.accelerate(50)
  carObject_1.accelerate(50)
  carObject_1.accelerate(50)

  carObject_1.decelerate(50)
  carObject_1.decelerate(50)
  carObject_1.decelerate(50)
  carObject_1.decelerate(50)

  carObject_2.accelerate(40)
  carObject_2.accelerate(40)
  carObject_2.__check_speed(500)

if __name__ == '__main__':
 main()
```

Abstraction

```
mpekmezci@cobalt:~/workspace/GTU-ENG-346-PRIVATE/object_oriented_programming-codes/02-abstraction$ python3 main.py
Accelerated current velocity is :50.1
Accelerated current velocity is :100.1
Already passed the maximum speed 120.
                Our speed is 100.1
Decelerated current velocity is :50.099999999999994
Decelerated current velocity is :0.09999999999999432
Speed is already 0.
Speed is already 0.
Accelerated current velocity is :40.1
Already passed the maximum speed 50.
                Our speed is 40.1
Traceback (most recent call last):
  File "/home/mpekmezci/workspace/GTU-ENG-346-PRIVATE/object_oriented_programming-codes/02-abstraction/main.py", line 23, in <module>
    main()
  File "/home/mpekmezci/workspace/GTU-ENG-346-PRIVATE/object_oriented_programming-codes/02-abstraction/main.py", line 20, in main
    carObject_2.__check_speed(500)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'CarClass' object has no attribute '__check_speed'
```

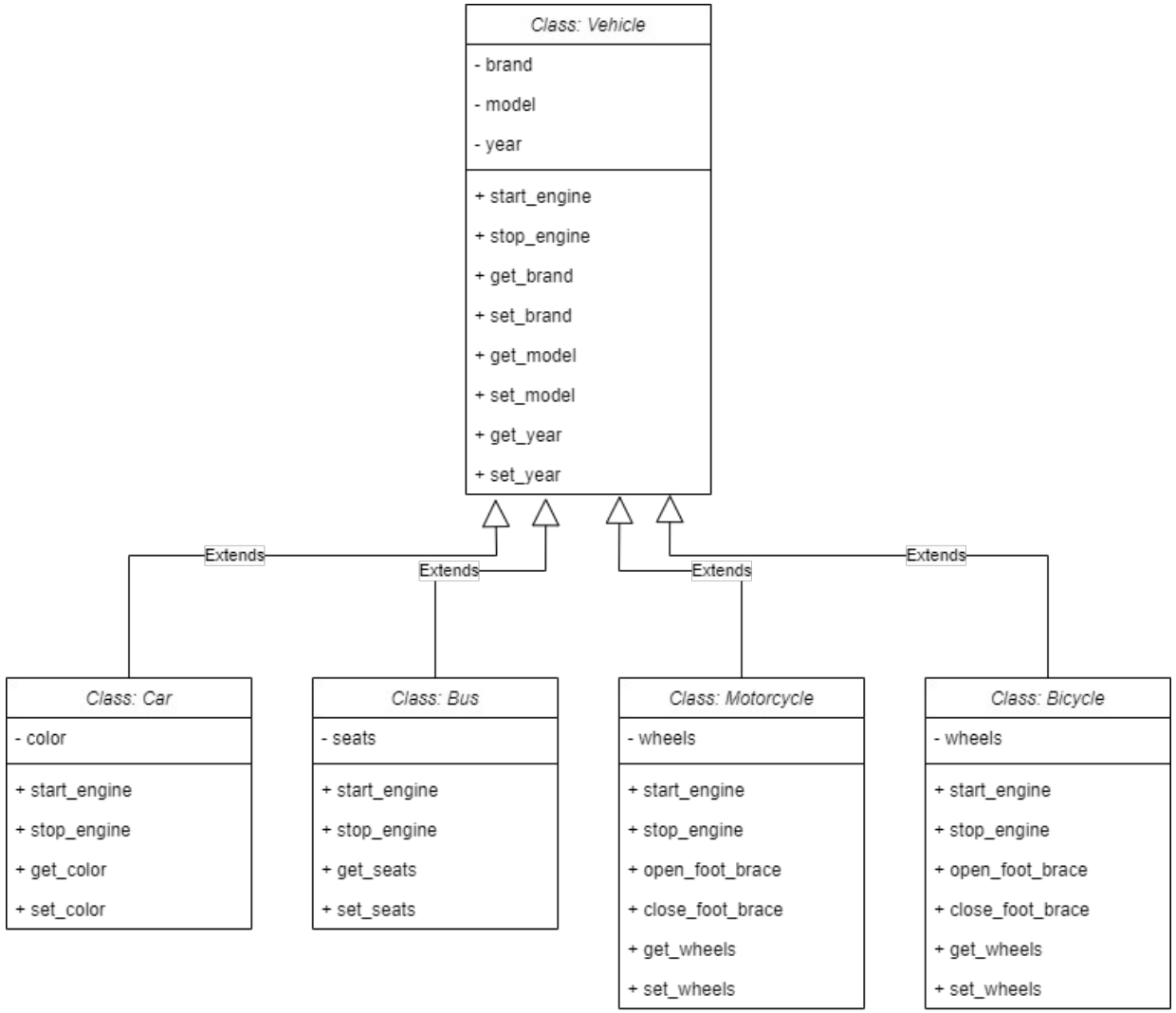
Encapsulation

- Concept of bundling data (attributes/properties) and methods (functions/behaviors) that operate on that data into a Class.
- Protects data from accidental modification.
- Prevents the developer to forget the update the code for related concepts.
- Enhances code organization.
- Streamlines interaction between program components.

Inheritance

- Hierarchical extension mechanism in OOP.
- The subclass inherits attributes and behaviors from the superclass and can extend or override them.
- Inheritance facilitates code reuse and allows you to model hierarchical relationships between classes.
- Importance: Inheritance promotes code reusability, as you can create new classes that inherit features from existing classes, reducing code duplication. It also simplifies the organization of related classes in a hierarchy, making it easier to understand and maintain the code.

Inheritance – Example



Polymorphism

- Subclasses are be treated as *instances* of a common superclass.
- It allows objects of different classes to be accessed and manipulated through a common (Class) interface, often through inheritance and method overriding.

- Importance: Polymorphism enhances flexibility and extensibility in OOP. It simplifies code by allowing you to write generic algorithms that can work with various objects without knowing their specific types. This makes it easier to adapt and extend software as new classes are added, leading to more maintainable and scalable code.

Polymorphism

```python
class Animal:
    def sound(self):
        return "Some generic sound"

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

class ANewAnimalType(Animal):
    def sound(self):
        return "A new animal type"

# Polymorphic behavior
animals = [Dog(), Cat(), ANewAnimalType(),Animal()]
for animal in animals:
    print(animal.sound())
```

```
mpekmezci@cobalt:~/workspace/GTU-ENG-346-PRIVATE/object_oriented_programming-codes/03-polymorphism$ python3 main.py
Bark
Meow
A new animal type
Some generic sound
mpekmezci@cobalt:~/workspace/GTU-ENG-346-PRIVATE/object_oriented_programming-codes/03-polymorphism$
```

Operator Overloading

- Overloading common operators (such as + operator) to extend the functionality of a (new) class.

- Example: Assume we are defining Complex Class and we want arithmetic operators to function on this new class.

Operator Overloading – continued

Common Syntax	Special Method Form	
a + b	a.__add__(b);	alternatively b.__radd__(a)
a − b	a.__sub__(b);	alternatively b.__rsub__(a)
a * b	a.__mul__(b);	alternatively b.__rmul__(a)
a / b	a.__truediv__(b);	alternatively b.__rtruediv__(a)
a // b	a.__floordiv__(b);	alternatively b.__rfloordiv__(a)
a % b	a.__mod__(b);	alternatively b.__rmod__(a)
a ** b	a.__pow__(b);	alternatively b.__rpow__(a)
a << b	a.__lshift__(b);	alternatively b.__rlshift__(a)
a >> b	a.__rshift__(b);	alternatively b.__rrshift__(a)
a & b	a.__and__(b);	alternatively b.__rand__(a)
a ^ b	a.__xor__(b);	alternatively b.__rxor__(a)
a | b	a.__or__(b);	alternatively b.__ror__(a)
a += b	a.__iadd__(b)	
a −= b	a.__isub__(b)	
a *= b	a.__imul__(b)	
...	...	

Non-Operator Overloads

- Overloading commonly used Python functions, such as str(), len(), etc.

v in a	a.__contains__(v)
a[k]	a.__getitem__(k)
a[k] = v	a.__setitem__(k,v)
del a[k]	a.__delitem__(k)
a(arg1, arg2, ...)	a.__call__(arg1, arg2, ...)
len(a)	a.__len__()
hash(a)	a.__hash__()
iter(a)	a.__iter__()
next(a)	a.__next__()
bool(a)	a.__bool__()
float(a)	a.__float__()
int(a)	a.__int__()
repr(a)	a.__repr__()
reversed(a)	a.__reversed__()
str(a)	a.__str__()

Named Variables in Function Definitions

- Example: Range Class

```
class RangeClass:
    def __init__ (self, start, stop=None, step=1):
```