Object-oriented programming - part 1

CMSC 104 Section 2
November 20, 2025

Administrative Notes

Classwork 9 is next Tuesday

Classwork 10 is going to be December 2 - the Tuesday after Thanksgiving.

- The syllabus is wrong; we will NOT have the classwork and quiz 4 on the same day

FINAL EXAM: Per https://registrar.umbc.edu/wp-content/uploads/sites/31/2025/09/Fall-2025-Regular-Final-Exam-Schedule.pdf

Our final is Thursday, December 11, from 6 to 8 pm, in the regular classroom.

Key Ideas of OOP

1. **Objects**
    - Self-contained units that bundle **data** (attributes) and **behavior** (methods).
2. **Classes**
    - Blueprints used to create objects.
    - Example: A `Car` class might define what all cars *are* and *can do*.
3. **Encapsulation**
    - Hiding internal details and exposing only what's necessary.
    - Helps keep code modular and secure.
4. **Inheritance**
    - A class can inherit attributes and methods from another class.
    - Example: `ElectricCar` inherits from `Car`.

Key Ideas of OOP (2 of 2)

5. Polymorphism

- Same interface, different behavior.
- Example: Different objects can implement their own version of a method like `drive()`.

6. Abstraction

- Simplifying complex systems by modeling only the essential aspects.

.

Why?

Why Use OOP?

- Makes code **modular**, **reusable**, and **easier to maintain**.
- Good for modeling complex systems.
- Common in languages like **Java, C++, Python, C#, Swift**, and others

If you're just writing a short, simple program to solve a specific problem, you're NOT going to use object-oriented programming

- There's too much overhead

But if you're writing complex code that's going to be used over and over

- **Especially if it's going to be used by multiple people**
- **Then OOP might just be the way to go!!**

Encapsulation

- Encapsulation bundles **data** (attributes) and **behaviors** (methods) into a single unit — a *class* — and restricts direct access to some of the object's internal details.

In Python, attributes can be:

- **Public**

  Accessible from anywhere.
  Example: `self.name`

- **Protected (not enforced; used by convention only)**

  Variable names start with a single underscore: `_balance`
  Indicates "internal use — don't touch from outside unless you know what you're doing."

- **Private attributes (implemented using name-mangling)**

  Variable names start with two underscores: `__balance`
  Python internally renames it to prevent direct access from outside the class

What’s “name mangling?”

When you have a variable inside a class definition

- And the variable name starts with a double underscore
- Python actually changes the variable name for its internal use to make sure somebody doesn’t “accidentally” use the private variable name and cause problems for your code

An example

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner                # public attribute
        self.__balance = balance          # private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
        else:    def get_balance(self):
            print("Insufficient funds")

   def get_balance(self):
        return self.__balance             # controlled access
```

```
acc = BankAccount("Alice", 100)
print(acc.__balance)  # AttributeError
```

If you try to directly access the private attribute, your attempt is rejected

```
print(acc.get_balance())
```

You can access the value through the defined interface - that's why this is "controlled access"

So why do you do this “encapsulation?”

It provides ***privacy*** and ***security***

- You do not let the user mess around directly with key data items
  - Passwords, bank account numbers, account balances, course grades
- You only expose sensitive data through interfaces that *you* control

Executive Order 14028 (May 2021; by President Biden)

- Requires “secure software development practices” for all software sold to/provided to Federal Government customers
- NIST Special Publication 800-218, Secure Software Development Framework

Do you care about this?

It depends on what kind of software you're going to write

- If you're just writing a program for yourself to solve some problem, you probably don't care
- If you're writing programs for others, you might well care
    - So OOP is a good thing to know

Polymorphism

In object-oriented programming, polymorphism is the provision of one interface to entities of different data types.[2] The concept is borrowed from a principle in biology in which an organism or species can have many different forms or stages.[3]

The most commonly recognized major forms of polymorphism are:

- *Ad hoc polymorphism*: defines a common interface for an arbitrary set of individually specified types.
- *Parametric polymorphism*: does not specify concrete types and instead uses abstract symbols that can substitute for any type.
- *Subtyping* (also called *subtype polymorphism* or *inclusion polymorphism*): when a name denotes instances of many different classes related by a common superclass.[4]

Source: https://en.wikipedia.org/wiki/Polymorphism_(computer_science)

What does this mean to us?

This is “ad hoc polymorphism”

First, define multiple Classes:

```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

class Duck:
    def speak(self):
        return "Quack!"
```

Next, define a function that uses polymorphism:

```
def animal sound(animal):
    print(animal.speak())
```

Finally, create objects and invoke the function:

```
# Create objects
dog = Dog()
cat = Cat()
duck = Duck()

# Same function, different behaviors
animal sound(dog)   # Woof!
animal sound(cat)   # Meow!
animal_sound(duck)  # Quack!
```

Subtype polymorphism

The parent class and each subclass have a different method called area

```
class Shape:
    def area(self):
        # Base implementation (optional)
        raise NotImplementedError("Subclasses must
implement area()")

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius * self.radius
```

```
def print_area(shape: Shape):
    # Subtype polymorphism: different
subclasses respond to area() differently
    print("Area:", shape.area())

# Subtype instances
r = Rectangle(4, 5)
c = Circle(3)

print_area(r)  # Area: 20
print_area(c)  # Area: 28.27431
```

In the previous slide:

The subclasses Circle and Rectangle each have a method, area()

- As does their parent class, Shape
- The subclass methods ***override*** the parent class method of the same name
- It is the subclass method that gets executed when operating on an object of the subclass type

But, why?

Why use polymorphism?

- It makes code more reusable
- It makes code somewhat simpler to understand
- It can make code smaller & more compact
  - Which may be important if you're talking tens of millions of lines of code
- It provides a layer of abstraction that may make your code more usable (& reusable) by and understandable to the layperson
  - It's another step in the progression:
    - Machine languages
    - Assembly languages
    - High-level languages

Who cares about abstraction?

We want the user to be able to use code to do a job; to do research; to play a game - to use it.

We do *not* want the user to have to be an expert in how the code is structured and why it is structured that way.