Lecture 24: Object Oriented Programming **Polymorphism**

Comp 102

Forman Christian University

Recap: Inheritance

Quick Recap: Class Hierarchy

```python
class Animal(object):
    def __init__(self, age):
        self.age = age
    def speak(self):
        print("some sound")

class Cat(Animal):
    def meo(self):
        print("meow")

class Dog(Animal):
    def bark(self):
        print("woof")
```

The Problem: Without Polymorphism

A Common Programming Problem

Suppose we want to make all our animals speak:

```python
animals = [Cat(2), Dog(3), Rabbit(1), Cat(5)]

# Without polymorphism, we'd need:
for animal in animals:
    if isinstance(animal, Cat):
        animal.meo()
    elif isinstance(animal, Dog):
        animal.bark()
    elif isinstance(animal, Rabbit):
        animal.squeak()
# Need to add elif for EVERY new animal type!
```

A Common Programming Problem

Suppose we want to make all our animals speak:

```
animals = [Cat(2), Dog(3), Rabbit(1), Cat(5)]

# Without polymorphism, we'd need:
for animal in animals:
    if isinstance(animal, Cat):
        animal.meo()
    elif isinstance(animal, Dog):
        animal.bark()
    elif isinstance(animal, Rabbit):
        animal.squeak()
# Need to add elif for EVERY new animal type!
```

Must check type of each animal manually - tedious!

A Common Programming Problem

Suppose we want to make all our animals speak:

```python
animals = [Cat(2), Dog(3), Rabbit(1), Cat(5)]

# Without polymorphism, we'd need:
for animal in animals:
    if isinstance(animal, Cat):
        animal.meo()
    elif isinstance(animal, Dog):
        animal.bark()
    elif isinstance(animal, Rabbit):
        animal.squeak()
# Need to add elif for EVERY new animal type!
```

Must check type of each animal manually - tedious!

This is tedious and doesn't scale!

The Solution

Polymorphism

"Many forms"

The ability to use objects of different types through a uniform interface

What is Polymorphism?

Polymorphism

- Greek: **"poly"** = many, **"morph"** = form

Polymorphism

- Greek: **"poly"** = many, **"morph"** = form
- The ability to **treat objects of different types** in a **similar way**

Polymorphism

- Greek: **"poly"** = many, **"morph"** = form
- The ability to **treat objects of different types** in a **similar way**
- Same **method name**, different **implementations**

Polymorphism

- Greek: **"poly"** = many, **"morph"** = form
- The ability to **treat objects of different types** in a **similar way**
- Same **method name**, different **implementations**
- Python automatically calls the **correct version** of the method based on the object's type

Polymorphism in Action

```python
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

animals = [Cat(2), Dog(3), Cat(1)]
for animal in animals:
    # Each calls their own version!
    animal.speak()
```

Polymorphism in Action

```python
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

animals = [Cat(2), Dog(3), Cat(1)]
for animal in animals:
    # Each calls their own version!
    animal.speak()
```

Same method name in all subclasses

Polymorphism in Action

```python
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

animals = [Cat(2), Dog(3), Cat(1)]
for animal in animals:
    # Each calls their own version!
    animal.speak()
```

Python automatically calls the correct speak() method!

Output

```
meow
woof
meow
```

Output

```
meow
woof
meow
```

- Python **automatically** determines which `speak()` to call
- Based on the **actual type** of the object
- We don't need to check types manually!

Why Use Polymorphism?

Benefits of Polymorphism

1. **Flexibility**
   - Write code that works with parent class but accepts any subclass

Benefits of Polymorphism

1. **Flexibility**
   - Write code that works with parent class but accepts any subclass
2. **Extensibility**
   - Add new subclasses without changing existing code

Benefits of Polymorphism

1. **Flexibility**
   - Write code that works with parent class but accepts any subclass
2. **Extensibility**
   - Add new subclasses without changing existing code
3. **Code Reusability**
   - One function works with many types

Benefits of Polymorphism

1. **Flexibility**
   - Write code that works with parent class but accepts any subclass
2. **Extensibility**
   - Add new subclasses without changing existing code
3. **Code Reusability**
   - One function works with many types
4. **Cleaner Code**
   - No need for long if-elif chains

Polymorphism Examples

Example 1: Animal Shelter

```python
def make_sound(animal):
    """Works with ANY Animal subclass"""
    animal.speak()

cat = Cat(2)
dog = Dog(3)
rabbit = Rabbit(1)

make_sound(cat)      # meow
make_sound(dog)      # woof
make_sound(rabbit)   # squeak
```

Example 1: Animal Shelter

```python
def make_sound(animal):
    """Works with ANY Animal subclass"""
    animal.speak()

cat = Cat(2)
dog = Dog(3)
rabbit = Rabbit(1)

make_sound(cat)      # meow
make_sound(dog)      # woof
make_sound(rabbit)   # squeak
```

Function accepts **Animal** but works with **any subclass**

Example 1: Animal Shelter

```python
def make_sound(animal):
    """Works with ANY Animal subclass"""
    animal.speak()

cat = Cat(2)
dog = Dog(3)
rabbit = Rabbit(1)

make_sound(cat)      # meow
make_sound(dog)      # woof
make_sound(rabbit)   # squeak
```

Same function, different behaviors!

Example 2: Processing Collections

```python
def morning_routine(animals):
    """Make all animals speak in the morning"""
    for animal in animals:
        animal.speak()

# Mix of different animal types
zoo = [Cat(2), Dog(3), Rabbit(1),
       Cat(1), Dog(5)]

morning_routine(zoo)
# Output: meow, woof, squeak, meow, woof
```

Example 3: More Complex Behavior

```python
class Animal:
    def __init__(self, age, name):
        self.age = age
        self.name = name
    def introduce(self):
        print(f"I'm {self.name}, I'm {self.age} years old")
        self.speak()

class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

c = Cat(2, "Fluffy")
c.introduce()  # I'm Fluffy, I'm 2 years old
               # meow
```

Example 3: More Complex Behavior

```python
class Animal:
    def __init__(self, age, name):
        self.age = age
        self.name = name
    def introduce(self):
        print(f"I'm {self.name}, I'm {self.age} years old")
        self.speak()

class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

c = Cat(2, "Fluffy")
c.introduce()  # I'm Fluffy, I'm 2 years old
               # meow
```

Parent method calls polymorphic method

Practice: You Try!

You Try! Exercise 1

Create a Shape hierarchy with polymorphic area() method:

```python
class Shape:
    def area(self):
        pass  # To be overridden

class Rectangle(Shape):
    def __init__(self, width, height):
        # Your code here
    def area(self):
        # Your code here

class Circle(Shape):
    def __init__(self, radius):
        # Your code here
    def area(self):
        # Your code here (use 3.14 for pi)
```

You Try! Exercise 1 (continued)

Write a function that uses polymorphism:

```python
def total_area(shapes):
    """
    Input: shapes is a list of Shape objects
    Returns: total area of all shapes
    """
    # Your code here
    pass

# Test your code:
shapes = [Rectangle(4, 5),
          Circle(3),
          Rectangle(2, 3)]
print(total_area(shapes))  # Should print: 54.26
```

You Try! Exercise 2

Create an `Employee` hierarchy:

```
class Employee:
    def __init__(self, name, base_salary):
        self.name = name
        self.base_salary = base_salary
    def calculate_pay(self):
        return self.base_salary

class Manager(Employee):
    def __init__(self, name, base_salary, bonus):
        # Your code: call parent __init__ and store bonus
    def calculate_pay(self):
        # Your code: return base_salary + bonus

class Salesperson(Employee):
    def __init__(self, name, base_salary, commission):
        # Your code: call parent __init__ and store commission
    def calculate_pay(self):
        # Your code: return base_salary + commission
```

You Try! Exercise 2 (continued)

```
def print_payroll(employees):
    """
    Input: employees is a list of Employee objects
    Prints: name and pay for each employee
    Returns: total payroll
    """
    # Your code here
    pass

# Test:
employees = [ Employee("Alice", 50000),  Manager("Bob", 60000, 10000),
    Salesperson("Charlie", 40000, 15000) ]

total = print_payroll(employees)
# Should print:
# Alice: \$50000
# Bob: \$70000
# Charlie: \$55000
# Total: \$175000
```

Abstract Base Classes

The Problem with Our Animal Class

- What if someone creates an `Animal` directly?

The Problem with Our Animal Class

- What if someone creates an `Animal` directly?
- `a = Animal(5, "Generic")`

The Problem with Our Animal Class

- What if someone creates an `Animal` directly?
- `a = Animal(5, "Generic")`
- What sound does a generic "animal" make?

The Problem with Our Animal Class

- What if someone creates an `Animal` directly?
- `a = Animal(5, "Generic")`
- What sound does a generic ”animal” make?
- We want `Animal` to be a **template** only

The Problem with Our Animal Class

- What if someone creates an `Animal` directly?
- `a = Animal(5, "Generic")`
- What sound does a generic "animal" make?
- We want `Animal` to be a **template** only
- Force subclasses to **implement** `speak()`

Solution

Abstract Base Class (ABC)

A class that:

- Cannot be instantiated directly
- Forces subclasses to implement certain methods
- Defines a **contract** for subclasses

Creating an Abstract Base Class

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    def __init__(self, age, name):
        self.age = age
        self.name = name

    @abstractmethod
    def speak(self):
        pass

    def introduce(self):
        print(f"I'm {self.name}, I'm {self.age} years old")
        self.speak()
```

Creating an Abstract Base Class

```
from abc import ABC, abstractmethod

class Animal(ABC):
    def __init__(self, age, name):
        self.age = age
        self.name = name

    @abstractmethod
    def speak(self):
        pass

    def introduce(self):
        print(f"I'm {self.name}, I'm {self.age} years old")
        self.speak()
```

Import ABC tools from Python

Creating an Abstract Base Class

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    def __init__(self, age, name):
        self.age = age
        self.name = name

    @abstractmethod
    def speak(self):
        pass

    def introduce(self):
        print(f"I'm {self.name}, I'm {self.age} years old")
        self.speak()
```

Inherit from ABC

Creating an Abstract Base Class

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    def __init__(self, age, name):
        self.age = age
        self.name = name

    @abstractmethod
    def speak(self):
        pass

    def introduce(self):
        print(f"I'm {self.name}, I'm {self.age} years old")
        self.speak()
```

Mark method as abstract - subclasses **must** implement it

Cannot Instantiate Abstract Classes

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

# This will cause an ERROR:
a = Animal(5, "Generic")

# TypeError: Can't instantiate abstract
# class Animal with abstract method speak
```

Cannot Instantiate Abstract Classes

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

# This will cause an ERROR:
a = Animal(5, "Generic")

# TypeError: Can't instantiate abstract
# class Animal with abstract method speak
```

Python prevents creating Animal objects!

Subclasses Must Implement Abstract Methods

```python
class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

# Now these work fine:
c = Cat(2, "Fluffy")
d = Dog(3, "Buddy")
c.speak()  # meow
d.speak()  # woof
```

Subclasses Must Implement Abstract Methods

```python
class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

# Now these work fine:
c = Cat(2, "Fluffy")
d = Dog(3, "Buddy")
c.speak()  # meow
d.speak()  # woof
```

Subclass implements the abstract method

Forgetting to Implement Causes Error

```python
class Rabbit(Animal):
    def hop(self):
        print("hopping")
    # Forgot to implement speak()!

# This will cause an ERROR:
r = Rabbit(1, "Fluffy")

# TypeError: Can't instantiate abstract
# class Rabbit with abstract method speak
```

Forgetting to Implement Causes Error

```python
class Rabbit(Animal):
    def hop(self):
        print("hopping")
    # Forgot to implement speak()!

# This will cause an ERROR:
r = Rabbit(1, "Fluffy")

# TypeError: Can't instantiate abstract
# class Rabbit with abstract method speak
```

Missing required speak() method

Forgetting to Implement Causes Error

```python
class Rabbit(Animal):
    def hop(self):
        print("hopping")
    # Forgot to implement speak()!

# This will cause an ERROR:
r = Rabbit(1, "Fluffy")

# TypeError: Can't instantiate abstract
# class Rabbit with abstract method speak
```

Python catches the error immediately!

Why Use Abstract Classes?

1. **Enforce consistency** - all subclasses have required methods
2. **Catch errors early** - at instantiation, not when method is called
3. **Document intent** - clearly shows which methods subclasses need
4. **Prevent misuse** - can't create incomplete objects

Common Pitfalls

Pitfall 1: Forgetting to Override

```python
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def meow(self):  # Wrong method name!
        print("meow")

c = Cat(2)
c.speak()  # Prints "some sound" (not "meow")
```

Pitfall 1: Forgetting to Override

```python
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def meow(self):  # Wrong method name!
        print("meow")

c = Cat(2)
c.speak()  # Prints "some sound" (not "meow")
```

Must use **same name** as parent method!

Pitfall 2: Wrong Method Signature

```python
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def speak(self, volume):  # Extra parameter!
        print(f"meow at volume {volume}")

def make_sound(animal):
    animal.speak()  # Error! Missing argument

c = Cat(2)
make_sound(c)  # TypeError!
```

Pitfall 2: Wrong Method Signature

```python
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def speak(self, volume):  # Extra parameter!
        print(f"meow at volume {volume}")

def make_sound(animal):
    animal.speak()  # Error! Missing argument

c = Cat(2)
make_sound(c)  # TypeError!
```

Signatures must match!

Key Principles

1. Use the **same method name** in parent and child
2. Keep the **same parameters** (method signature)
3. Write functions that accept **parent type** but work with **any subclass**
4. Python handles the rest **automatically**!

Real-World Application

Real-World Example: Payment Processing

```python
class PaymentMethod(ABC): # Abstract base class
    @abstractmethod
    def process(self, amount):
        pass

class CreditCard(PaymentMethod):
    def process(self, amount):
        print(f"Charging {amount} to card")

class PayPal(PaymentMethod):
    def process(self, amount):
        print(f"Charging {amount} to PayPal")

class Cash(PaymentMethod):
    def process(self, amount):
        print(f"Receiving {amount} in cash")
```

Real-World Example: Payment Processing (continued)

```python
def checkout(total, payment_method):
    """Works with ANY payment method"""
    print(f"Total: {total}")
    payment_method.process(total)

# Client code doesn't care about payment type!
cc = CreditCard()
pp = PayPal()
cash = Cash()

checkout(100, cc)     # Charging 100 to card
checkout(200, pp)     # Charging 200 to PayPal
checkout(50, cash)    # Receiving 50 in cash
```

Real-World Example: Payment Processing (continued)

```python
def checkout(total, payment_method):
    """Works with ANY payment method"""
    print(f"Total: {total}")
    payment_method.process(total)

# Client code doesn't care about payment type!
cc = CreditCard()
pp = PayPal()
cash = Cash()

checkout(100, cc)     # Charging 100 to card
checkout(200, pp)     # Charging 200 to PayPal
checkout(50, cash)    # Receiving 50 in cash
```

Single function works with all payment types!

Real-World Example: Payment Processing (continued)

```python
def checkout(total, payment_method):
    """Works with ANY payment method"""
    print(f"Total: {total}")
    payment_method.process(total)

# Client code doesn't care about payment type!
cc = CreditCard()
pp = PayPal()
cash = Cash()

checkout(100, cc)     # Charging 100 to card
checkout(200, pp)     # Charging 200 to PayPal
checkout(50, cash)    # Receiving 50 in cash
```

Polymorphism in action!

Duck Typing in Python

Duck Typing

- Python's approach to polymorphism is **"duck typing"**

Duck Typing

- Python's approach to polymorphism is **"duck typing"**
- *"If it walks like a duck and quacks like a duck, then it must be a duck"*

Duck Typing

- Python's approach to polymorphism is **"duck typing"**
- *"If it walks like a duck and quacks like a duck, then it must be a duck"*
- Python doesn't care about the **type** of an object

Duck Typing

- Python's approach to polymorphism is **"duck typing"**
- *"If it walks like a duck and quacks like a duck, then it must be a duck"*
- Python doesn't care about the **type** of an object
- Python only cares if the object has the **right methods**

Duck Typing

- Python's approach to polymorphism is **"duck typing"**
- *"If it walks like a duck and quacks like a duck, then it must be a duck"*
- Python doesn't care about the **type** of an object
- Python only cares if the object has the **right methods**
- Objects don't even need to inherit from the same parent!

Duck Typing Example

```python
class Dog:
    def speak(self):
        print("woof")

class Robot:  # Not related to Animal!
    def speak(self):
        print("beep boop")

class Person:
    def speak(self):
        print("hello")

def make_speak(thing):
    thing.speak()  # Works with anything that has speak()

make_speak(Dog())      # woof
make_speak(Robot())    # beep boop
make_speak(Person())   # hello
```

Duck Typing Example

```
class Dog:
    def speak(self):
        print("woof")

class Robot:  # Not related to Animal!
    def speak(self):
        print("beep boop")

class Person:
    def speak(self):
        print("hello")

def make_speak(thing):
    thing.speak()  # Works with anything that has speak()

make_speak(Dog())     # woof
make_speak(Robot())   # beep boop
make_speak(Person())  # hello
```

No inheritance relationship needed!

Duck Typing Example

```python
class Dog:
    def speak(self):
        print("woof")

class Robot:  # Not related to Animal!
    def speak(self):
        print("beep boop")

class Person:
    def speak(self):
        print("hello")

def make_speak(thing):
    thing.speak()  # Works with anything that has speak()

make_speak(Dog())      # woof
make_speak(Robot())    # beep boop
make_speak(Person())   # hello
```

Works with **any** object with a `speak()` method

Advanced: Method Resolution Order

How Python Finds Methods

When you call `object.method()`:

1. Look in the **object's class**

How Python Finds Methods

When you call `object.method()`:

1. Look in the **object's class**
2. If not found, look in the **parent class**

How Python Finds Methods

When you call `object.method()`:

1. Look in the **object's class**
2. If not found, look in the **parent class**
3. Continue up the **inheritance hierarchy**

How Python Finds Methods

When you call `object.method()`:

1. Look in the **object's class**
2. If not found, look in the **parent class**
3. Continue up the **inheritance hierarchy**
4. Use the **first match** found

How Python Finds Methods

When you call `object.method()`:

1. Look in the **object's class**
2. If not found, look in the **parent class**
3. Continue up the **inheritance hierarchy**
4. Use the **first match** found

This is called **Method Resolution Order (MRO)**

Visualizing Method Resolution

```
class Animal(ABC):
    @abstractmethod
    def speak(self):
        print("some sound")
    def eat(self):
        print("eating")

class Cat(Animal):
    def speak(self):  # Overrides Animal.speak()
        print("meow")
    # eat() is inherited, not overridden

c = Cat(5)
c.speak()   # Calls Cat.speak() - found in Cat
c.eat()     # Calls Animal.eat() - not in Cat,
            # searches up to Animal
```

Visualizing Method Resolution

```python
class Animal(ABC):
    @abstractmethod
    def speak(self):
        print("some sound")
    def eat(self):
        print("eating")

class Cat(Animal):
    def speak(self):  # Overrides Animal.speak()
        print("meow")
    # eat() is inherited, not overridden

c = Cat(5)
c.speak()    # Calls Cat.speak() - found in Cat
c.eat()      # Calls Animal.eat() - not in Cat,
             # searches up to Animal
```

Found in Cat class - stop searching

Visualizing Method Resolution

```
class Animal(ABC):
    @abstractmethod
    def speak(self):
        print("some sound")
    def eat(self):
        print("eating")

class Cat(Animal):
    def speak(self):  # Overrides Animal.speak()
        print("meow")
    # eat() is inherited, not overridden

c = Cat(5)
c.speak()  # Calls Cat.speak() - found in Cat
c.eat()    # Calls Animal.eat() - not in Cat,
           # searches up to Animal
```

Not in Cat, search parent Animal

Polymorphism vs Method Overriding

Key Distinction

- **Method Overriding**: Defining a method in a child class with the same name as in the parent
- **Polymorphism**: The ability to use overridden methods through a common interface
- Overriding is the **mechanism**
- Polymorphism is the **result**

Overriding vs Polymorphism

```python
class Animal(ABC):
    @abstractmethod
    def speak(self):
        print("some sound")

class Cat(Animal):
    def speak(self):
        print("meow")

def make_sound(animal):
    animal.speak()

c = Cat(2)
make_sound(c)  # Polymorphism: function works with
               # any Animal, calls right speak()
```

Overriding vs Polymorphism

```python
class Animal(ABC):
    @abstractmethod
    def speak(self):
        print("some sound")

class Cat(Animal):
    def speak(self):
        print("meow")

def make_sound(animal):
    animal.speak()

c = Cat(2)
make_sound(c)  # Polymorphism: function works with
               # any Animal, calls right speak()
```

Overriding: defining the method

Overriding vs Polymorphism

```python
class Animal(ABC):
    @abstractmethod
    def speak(self):
        print("some sound")

class Cat(Animal):
    def speak(self):
        print("meow")

def make_sound(animal):
    animal.speak()

c = Cat(2)
make_sound(c)  # Polymorphism: function works with
               # any Animal, calls right speak()
```

Polymorphism: using the overridden methods

Summary

Summary: Key Takeaways

1. **Polymorphism** = “many forms”
2. Write code that works with **parent class**, automatically works with **all subclasses**
3. Same **method name**, different **implementations**
4. Python uses **duck typing** - only cares about methods, not types
5. Makes code more **flexible**, **extensible**, and **reusable**

Remember

Polymorphism allows you to write functions that work with **many different types** of objects through a **common interface**

This is one of the most powerful features of OOP!

Questions?