Report: Object-Oriented Programming
Polymorphism and Duck Typing

Nguyễn Tuấn Đức

February 27, 2026

1 Polymorphism

1.1 Definition

Polymorphism, derived from the Greek words "poly" (many) and "morph" (forms), is a fundamental concept in Object-Oriented Programming (OOP). It refers to the ability of different objects to respond to the same method call in a way that is specific to their type. In essence, it allows a single interface to represent different underlying forms, whether they are different data types or classes.

1.2 Application

Polymorphism is widely used to design highly extensible and maintainable software architectures. It enables developers to write generic code that can operate on objects of various classes without needing to know their exact types. Common applications include implementing callback functions, designing robust APIs, and creating collections of heterogeneous objects that share a common interface.

1.3 Code Simulation

Below is a Python example demonstrating polymorphism through method overriding. The `animal_sound` function acts as a unified interface for different animal types.

```
class Animal:
    def speak(self):
        pass

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

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

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

```
15
16 dog = Dog()
17 cat = Cat()
18
19 animal_sound(dog)
20 animal_sound(cat)
```

2 Duck Typing

2.1 Definition

Duck typing is a concept related to dynamic typing, famously summarized by the phrase: "If it walks like a duck and quacks like a duck, then it must be a duck." In programming, this means that the suitability of an object is determined solely by the presence of specific methods and properties, rather than the explicit type or inheritance hierarchy of the object itself.

2.2 Application

Duck typing is extensively applied in dynamically typed languages to maximize flexibility. It allows functions and methods to accept any object as long as it provides the required behavior, completely bypassing strict inheritance trees. This approach is highly useful in building adaptable plugins, processing various data streams seamlessly, and reducing the boilerplate code typically associated with rigid interfaces.

2.3 Code Simulation

This Python snippet illustrates duck typing. The `make_it_fly` function does not check if the object belongs to a specific class; it only cares whether the object has a `fly` method.

```
1  class Bird:
2      def fly(self):
3          print("Flapping wings to fly.")
4
5  class Airplane:
6      def fly(self):
7          print("Starting engines to fly.")
8
9  def make_it_fly(flying_entity):
10     flying_entity.fly()
11
12 bird = Bird()
13 plane = Airplane()
14
15 make_it_fly(bird)
16 make_it_fly(plane)
```