

## Лекция 7

### Полиморфизм: динамическое и статическое связывание

#### 1. Введение в полиморфизм

Полиморфизм является одним из ключевых принципов объектно-ориентированного программирования (ООП) и предоставляет возможность объектам разного типа реагировать на одно и то же сообщение (вызов метода) по-разному. Дословно «полиморфизм» означает «многообразие форм», и этот принцип позволяет использовать один интерфейс для работы с разными типами объектов, при этом каждый объект может реализовать этот интерфейс уникальным образом.

Полиморфизм делает программы более гибкими и позволяет снизить избыточность кода, так как методы могут быть вызваны для объектов различных классов, даже если их поведение зависит от конкретного типа объекта. В ООП полиморфизм можно реализовать через наследование и интерфейсы, что позволяет изменять или расширять функциональность программных систем, не изменяя их базовую структуру.

Цель этой лекции — объяснить принципы полиморфизма, рассмотреть его виды, такие как статическое и динамическое связывание, а также продемонстрировать примеры использования полиморфизма в программировании.

#### 2. Основы полиморфизма

Полиморфизм позволяет методам класса или интерфейса работать с разными типами данных или выполнять разные действия, в зависимости от типа данных, с которыми они работают. Полиморфизм можно разделить на два основных типа:

- **Компиляционный (статический) полиморфизм** — когда связывание метода с вызовом происходит на этапе компиляции.
- **Исполнительный (динамический) полиморфизм** — когда связывание метода с вызовом происходит во время выполнения программы.

##### 2.1 Примеры полиморфизма

Полиморфизм может проявляться в разных формах, таких как переопределение методов и перегрузка методов. Рассмотрим пример, когда метод `draw()` вызывается для разных объектов и вызывает различные реализации:

```
python
Копировать код
class Shape:
    def draw(self):
        raise NotImplementedError("Subclass must implement abstract method")

class Circle(Shape):
    def draw(self):
        print("Drawing a circle")

class Square(Shape):
    def draw(self):
        print("Drawing a square")

shapes = [Circle(), Square()]

for shape in shapes:
    shape.draw()
```

Здесь метод `draw` вызывается для объектов классов `Circle` и `Square`, и каждый объект реализует метод по-своему, демонстрируя принцип полиморфизма.

### 3. Статическое связывание

Статическое связывание, также известное как раннее связывание, происходит на этапе компиляции. Это означает, что компилятор определяет, какой метод следует вызвать, исходя из типа переменной. Статическое связывание применимо в случаях, когда компилятор может точно определить, какой метод будет вызван.

#### 3.1 Перегрузка методов

Перегрузка методов — это пример статического полиморфизма. Она позволяет создавать несколько методов с одинаковым именем в одном классе, но с разными параметрами. Компилятор на этапе компиляции определяет, какой из методов следует вызвать, исходя из переданных параметров.

Пример перегрузки метода:

```
python
Копировать код
class Printer:
    def print_data(self, data):
        print("Printing:", data)

    def print_data(self, data, count):
```

```
for _ in range(count):  
    print("Printing:", data)
```

```
# Использование  
printer = Printer()  
printer.print_data("Hello", 3) # вызовет метод с двумя аргументами
```

В некоторых языках, таких как Java или C++, перегрузка методов возможна, но в Python перегрузка методов реализована неявно, поскольку последний определенный метод будет заменять предыдущий.

### 3.2 Перегрузка операторов

Перегрузка операторов — это еще один пример статического полиморфизма, позволяющий определять, как операторы, такие как +, -, \*, должны работать с пользовательскими типами данных. Перегрузка операторов позволяет использовать привычные операции для объектов пользовательских классов.

Пример перегрузки оператора:

```
python  
Копировать код  
class Vector:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __add__(self, other):  
        return Vector(self.x + other.x, self.y + other.y)  
  
    def __str__(self):  
        return f"Vector({self.x}, {self.y})"  
  
v1 = Vector(2, 3)  
v2 = Vector(4, 5)  
print(v1 + v2) # Вывод: Vector(6, 8)
```

Здесь перегружен оператор +, который позволяет складывать два объекта типа Vector.

### 4. Динамическое связывание

Динамическое связывание, или позднее связывание, — это полиморфизм, при котором определение вызываемого метода происходит на этапе выполнения. Этот тип полиморфизма позволяет создавать более гибкие и расширяемые

программы, так как метод, который будет вызван, определяется только в момент выполнения, исходя из фактического типа объекта.

#### 4.1 Переопределение методов

Переопределение методов — это процесс, при котором подкласс предоставляет собственную реализацию метода, уже определенного в суперклассе. Динамическое связывание позволяет программе выбрать нужный метод во время выполнения в зависимости от фактического типа объекта.

Пример переопределения метода:

```
python
Копировать код
class Animal:
    def make_sound(self):
        print("Animal sound")

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")

animals = [Dog(), Cat()]

for animal in animals:
    animal.make_sound()
```

В этом примере метод `make_sound` переопределяется в каждом подклассе, и вызывается версия метода, соответствующая типу объекта, демонстрируя динамическое связывание.

#### 4.2 Динамическое связывание и абстрактные классы

Абстрактные классы — это классы, которые не могут быть созданы как объекты и служат для определения интерфейса для подклассов. Абстрактные методы, определенные в абстрактных классах, должны быть переопределены в подклассах. Это обеспечивает полиморфизм, так как позволяет вызывать методы подклассов через ссылки на абстрактный суперкласс.

Пример использования абстрактного класса:

```
python
Копировать код
from abc import ABC, abstractmethod
```

```
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

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

    def area(self):
        return 3.1415 * self.radius ** 2

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

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

shapes = [Circle(5), Rectangle(4, 6)]

for shape in shapes:
    print("Area:", shape.area())
```

Здесь класс Shape — абстрактный, и каждый подкласс реализует метод area по-своему, демонстрируя динамическое связывание.

## 5. Преимущества и недостатки полиморфизма

### 5.1 Преимущества полиморфизма

- **Гибкость:** Полиморфизм позволяет разработчикам писать код, который может работать с объектами разных типов, обеспечивая гибкость и расширяемость программ.
- **Упрощение кода:** Полиморфизм помогает упростить код за счет сокращения количества проверок типов и структур if-else, так как методы могут быть вызваны для любого типа объектов, реализующих интерфейс.

- **Расширяемость:** Система становится более расширяемой, так как добавление новых типов объектов не требует изменений в существующем коде, который использует полиморфные методы.

## 5.2 Недостатки полиморфизма

- **Сложность отладки:** Динамическое связывание может усложнить отладку, так как трудно отследить, какой метод будет вызван на этапе выполнения.
- **Потенциальная неэффективность:** Динамическое связывание требует времени на определение фактического типа объекта на этапе выполнения, что может снизить производительность.
- **Риск ошибок:** Полиморфизм может привести к ошибкам, если методы разных объектов имеют несовместимое поведение, что может вызвать неожиданные результаты.

## 6. Примеры полиморфизма в реальных приложениях

Полиморфизм широко используется в различных приложениях для обработки объектов разных типов единым способом. Рассмотрим несколько примеров:

### 6.1 Полиморфизм в графических интерфейсах

В графических интерфейсах полиморфизм позволяет обрабатывать различные типы виджетов (кнопки, текстовые поля, изображения) единым образом. Например, каждый виджет может иметь метод `render()`, но реализация метода будет различаться в зависимости от типа виджета.

```
python
Копировать код
class Widget(ABC):
    @abstractmethod
    def render(self):
        pass

class Button(Widget):
    def render(self):
        return "Rendering a button"

class TextField(Widget):
    def render(self):
        return "Rendering a text field"

widgets = [Button(), TextField()]

for widget in widgets:
```

```
print(widget.render())
```

## **6.2 Полиморфизм в обработке данных**

Полиморфизм полезен при обработке различных типов данных. Например, классы `CSVData`, `JSONData` и `XMLData` могут реализовать метод `parse()`, и программа может обрабатывать любой формат данных через единый интерфейс.

## **7. Заключение**

Полиморфизм — это мощный принцип объектно-ориентированного программирования, который обеспечивает высокую гибкость и расширяемость кода. Он позволяет объектам разных классов обрабатывать вызовы через единый интерфейс, адаптируя своё поведение в зависимости от конкретного типа. Благодаря полиморфизму разработчики могут создавать универсальные решения, которые легко расширяются и адаптируются к новым требованиям без необходимости значительных изменений в коде. Этот принцип поддерживает модульность и масштабируемость, делая программное обеспечение более устойчивым и легко поддающимся развитию.